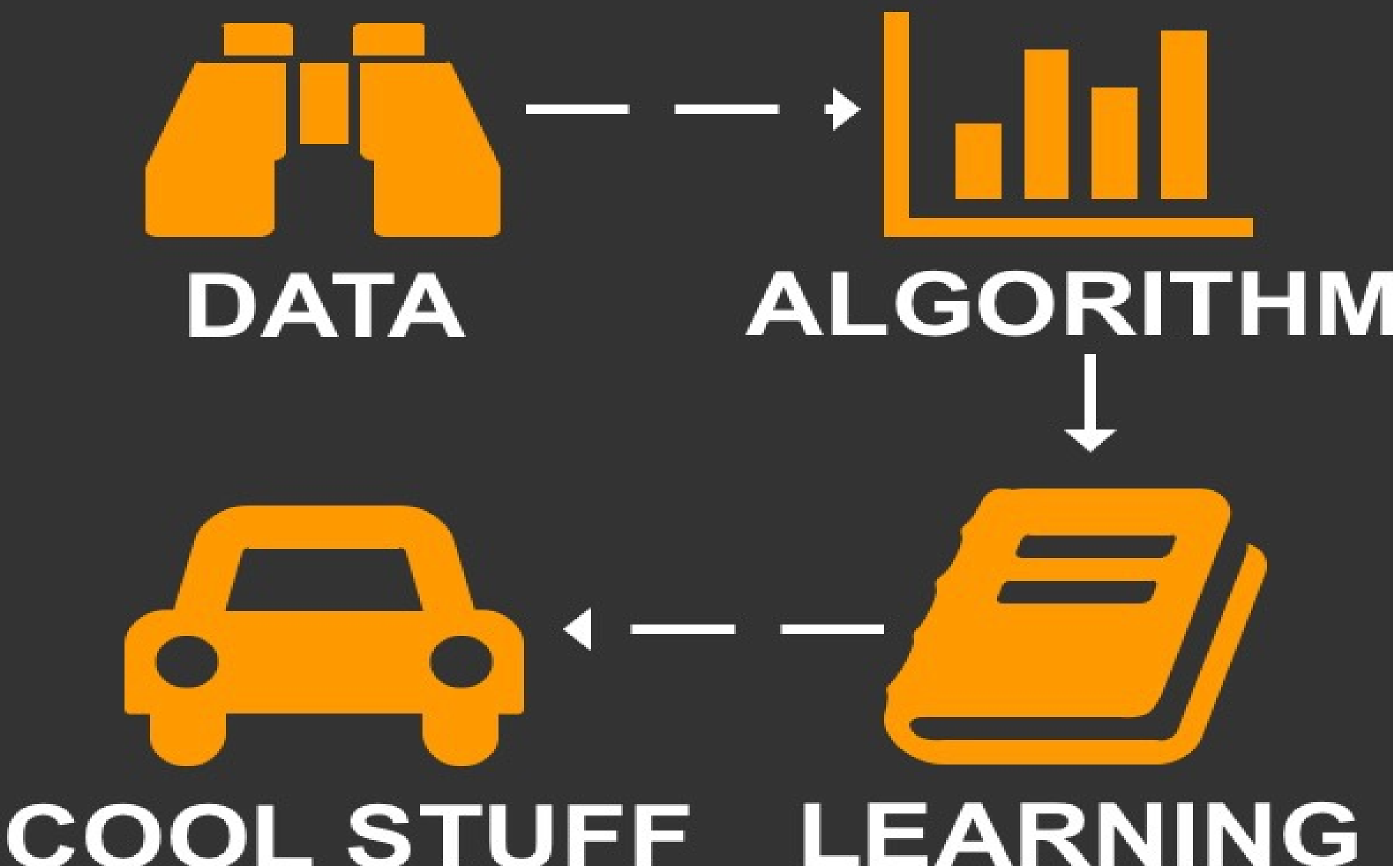
MACHINE LEARNING

For Absolute Beginners

Second Edition



Oliver Theobald

Machine Learning For Absolute Beginners

Oliver Theobald

Second Edition

Contents

INTRODUCTION

Machines have come a long way since the Industrial Revolution. They continue to fill factory floors and manufacturing plants, but now their capabilities extend beyond manual activities to cognitive tasks that, until recently, only humans were capable of performing. Judging song competitions, driving automobiles, and mopping the floor with professional chess players are three examples of the specific complex tasks machines are now capable of simulating.

But their remarkable feats trigger fear among some observers. Part of this fear nestles on the neck of survivalist insecurities, where it provokes the deep-seated question of *what if*? *What if* intelligent machines turn on us in a struggle of the fittest? *What if* intelligent machines produce offspring with capabilities that humans never intended to impart to machines? *What if* the legend of the *singularity* is true?

The other notable fear is the threat to job security, and if you're a truck driver or an accountant, there is a valid reason to be worried. According to the British Broadcasting Company's (BBC) interactive online resource *Will a robot take my job?*, professions such as bar worker (77%), waiter (90%), chartered accountant (95%), receptionist (96%), and taxi driver (57%) each have a high chance of becoming automated by the year 2035.[1]

But research on planned job automation and crystal ball gazing with respect to the future evolution of machines and artificial intelligence (AI) should be read with a pinch of skepticism. AI technology is moving fast, but broad adoption is still an unchartered path fraught with known and unforeseen challenges. Delays and other obstacles are inevitable.

Nor is machine learning a simple case of flicking a switch and asking the machine to predict the outcome of the Super Bowl and serve you a delicious martini. Machine learning is far from what you would call an out-of-the-box solution.

Machines operate based on statistical algorithms managed and overseen by skilled individuals—known as *data scientists* and *machine learning engineers*. This is one labor market where job opportunities are destined for

growth but where, currently, supply is struggling to meet demand. Industry experts lament that one of the biggest obstacles delaying the progress of AI is the inadequate supply of professionals with the necessary expertise and training.

According to Charles Green, the Director of Thought Leadership at Belatrix Software:

> *"It's a huge challenge to find data scientists, people with machine learning experience, or people with the skills to analyze and use the data, as well as those who can create the algorithms required for machine learning. Secondly, while the technology is still emerging, there are many ongoing developments. It's clear that AI is a long way from how we might imagine it."* [2]

Perhaps your own path to becoming an expert in the field of machine learning starts here, or maybe a baseline understanding is sufficient to satisfy your curiosity for now. In any case, let's proceed with the assumption that you are receptive to the idea of training to become a successful data scientist or machine learning engineer.

To build and program intelligent machines, you must first understand classical statistics. Algorithms derived from classical statistics contribute the metaphorical blood cells and oxygen that power machine learning. Layer upon layer of linear regression, *k*-nearest neighbors, and random forests surge through the machine and drive their cognitive abilities. Classical statistics is at the heart of machine learning and many of these algorithms are based on the same statistical equations you studied in high school. Indeed, statistical algorithms were conducted on paper well before machines ever took on the title of *artificial intelligence*.

Computer programming is another indispensable part of machine learning. There isn't a click-and-drag or Web 2.0 solution to perform advanced machine learning in the way one can conveniently build a website nowadays with WordPress or Strikingly. Programming skills are therefore vital to manage data and design statistical models that run on machines.

Some students of machine learning will have years of programming experience but haven't touched classical statistics since high school. Others, perhaps, never even attempted statistics in their high school years. But not to worry, many of the machine learning algorithms we discuss in this book have working implementations in your programming language of choice; no equation writing necessary. You can use code to execute the actual number

crunching for you.

If you have not learned to code before, you will need to if you wish to make further progress in this field. But for the purpose of this compact starter's course, the curriculum can be completed without any background in computer programming. This book focuses on the high-level fundamentals of machine learning as well as the mathematical and statistical underpinnings of designing machine learning models.

For those who do wish to look at the programming aspect of machine learning, Chapter 13 walks you through the entire process of setting up a supervised learning model using the popular programming language Python.

WHAT IS MACHINE LEARNING?

In 1959, IBM published a paper in the *IBM Journal of Research and Development* with an, at the time, obscure and curious title. Authored by IBM's Arthur Samuel, the paper invested the use of machine learning in the game of checkers "to verify the fact that a computer can be programmed so that it will learn to play a better game of checkers than can be played by the person who wrote the program."[3]

Although it was not the first publication to use the term "machine learning" per se, Arthur Samuel is widely considered as the first person to coin and define machine learning in the form we now know today. Samuel's landmark journal submission, *Some Studies in Machine Learning Using the Game of Checkers,* is also an early indication of homo sapiens' determination to impart our own system of learning to man-made machines.



Figure 1: Historical mentions of "machine learning" in published books. *Source: Google Ngram Viewer, 2017*

Arthur Samuel introduces machine learning in his paper as a subfield of computer science that gives computers the ability to learn without being explicitly programmed. [4] Almost six decades later, this definition remains widely accepted.

Although not directly mentioned in Arthur Samuel's definition, a key feature of machine learning is the concept of *self-learning*. This refers to the application of statistical modeling to detect patterns and improve

performance based on data and empirical information; all without direct programming commands. This is what Arthur Samuel described as the ability to learn without being explicitly programmed. But he doesn't infer that machines formulate decisions with no upfront programming. On the contrary, machine learning is heavily dependent on computer programming. Instead, Samuel observed that machines don't require a direct *input command* to perform a set task but rather *input data*.

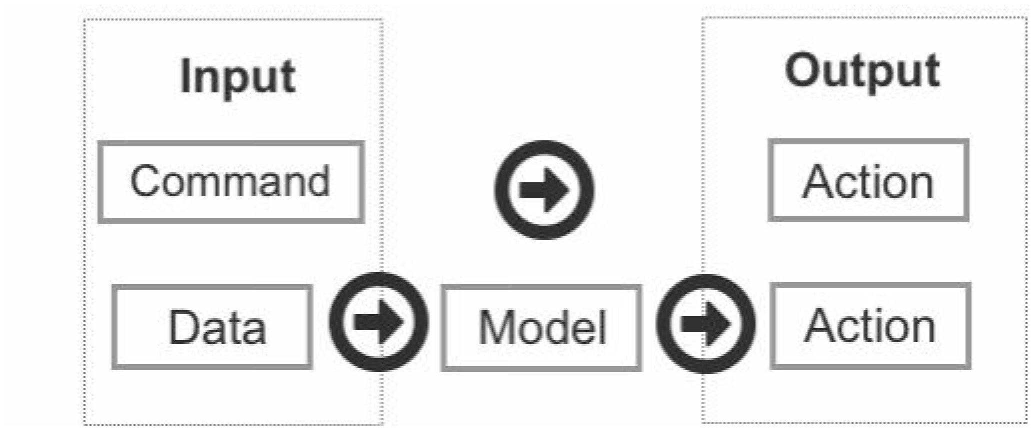


Figure 2: Comparison of Input Command vs Input Data

An example of an input command is typing "2+2" into a programming language such as Python and hitting "Enter."

```
>>> 2+2
4
>>>
```

This represents a direct command with a direct answer.

Input data, however, is different. Data is fed to the machine, an algorithm is selected, hyperparameters (settings) are configured and adjusted, and the machine is instructed to conduct its analysis. The machine proceeds to decipher patterns found in the data through the process of trial and error. The machine's data model, formed from analyzing data patterns, can then be used to predict future values.

Although there is a relationship between the programmer and the machine, they operate a layer apart in comparison to traditional computer programming. This is because the machine is formulating decisions based on experience and mimicking the process of human-based decision-making.

As an example, let's say that after examining the YouTube viewing habits of data scientists your machine identifies a strong relationship between data

scientists and cat videos. Later, your machine identifies patterns among the physical traits of baseball players and their likelihood of winning the season's Most Valuable Player (MVP) award. In the first scenario, the machine analyzed what videos data scientists enjoy watching on YouTube based on user engagement; measured in likes, subscribes, and repeat viewing. In the second scenario, the machine assessed the physical features of previous baseball MVPs among various other features such as age and education. However, in neither of these two scenarios was your machine explicitly programmed to produce a direct outcome. You fed the input data and configured the nominated algorithms, but the final prediction was determined by the machine through self-learning and data modeling.

You can think of building a data model as similar to training a guide dog. Through specialized training, guide dogs learn how to respond in various situations. For example, the dog will learn to heel at a red light or to safely lead its master around obstacles. If the dog has been properly trained, then, eventually, the trainer will no longer be required; the guide dog will be able to apply its training in various unsupervised situations. Similarly, machine learning models can be trained to form decisions based on past experience.

A simple example is creating a model that detects spam email messages. The model is trained to block emails with suspicious subject lines and body text containing three or more flagged keywords: dear friend, free, invoice, PayPal, Viagra, casino, payment, bankruptcy, and winner. At this stage, though, we are not yet performing machine learning. If we recall the visual representation of *input command vs input data*, we can see that this process consists of only two steps: Command > Action.

Machine learning entails a three-step process: Data > Model > Action.

Thus, to incorporate machine learning into our spam detection system, we need to switch out "command" for "data" and add "model" in order to produce an action (output). In this example, the data comprises sample emails and the model consists of statistical-based rules. The parameters of the model include the same keywords from our original negative list. The model is then trained and tested against the data.

Once the data is fed into the model, there is a strong chance that assumptions contained in the model will lead to some inaccurate predictions. For example, under the rules of this model, the following email subject line would automatically be classified as spam: "**PayPal** has received your **payment** for **Casino** Royale purchased on eBay."

As this is a genuine email sent from a PayPal auto-responder, the spam detection system is lured into producing a false positive based on the negative list of keywords contained in the model. Traditional programming is highly susceptible to such cases because there is no built-in mechanism to test assumptions and modify the rules of the model. Machine learning, on the other hand, can adapt and modify assumptions through its three-step process and by reacting to errors.

Training & Test Data

In machine learning, data is split into *training data* and *test data*. The first split of data, i.e. the initial reserve of data you use to develop your model, provides the training data. In the spam email detection example, false positives similar to the PayPal auto-response might be detected from the training data. New rules or modifications must then be added, e.g., email notifications issued from the sending address "payments@paypal.com" should be excluded from spam filtering.

After you have successfully developed a model based on the training data and are satisfied with its accuracy, you can then test the model on the remaining data, known as the test data. Once you are satisfied with the results of both the training data and test data, the machine learning model is ready to filter incoming emails and generate decisions on how to categorize those incoming messages.

The difference between machine learning and traditional programming may seem trivial at first, but it will become clear as you run through further examples and witness the special power of self-learning in more nuanced situations.

The second important point to take away from this chapter is how machine learning fits into the broader landscape of data science and computer science. This means understanding how machine learning interrelates with parent fields and sister disciplines. This is important, as you will encounter these related terms when searching for relevant study materials—and you will hear them mentioned ad nauseam in introductory machine learning courses. Relevant disciplines can also be difficult to tell apart at first glance, such as "machine learning" and "data mining."

Let's begin with a high-level introduction. Machine learning, data mining, computer programming, and most relevant fields (excluding classical

statistics) derive first from computer science, which encompasses everything related to the design and use of computers. Within the all-encompassing space of computer science is the next broad field: data science. Narrower than computer science, data science comprises methods and systems to extract knowledge and insights from data through the use of computers.

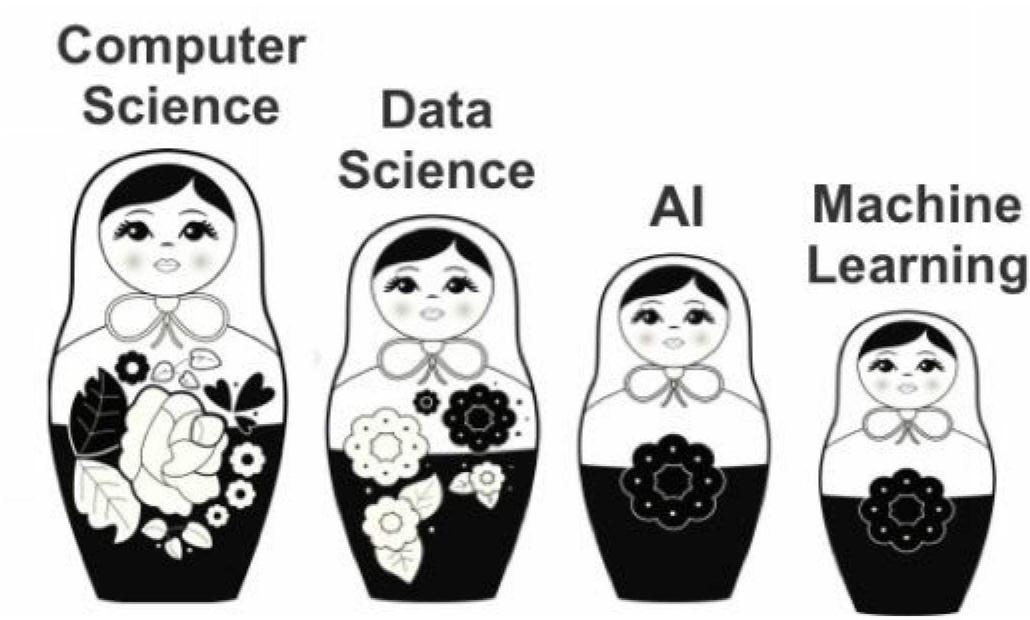


Figure 3: The lineage of machine learning represented by a row of Russian matryoshka dolls

Popping out from computer science and data science as the third matryoshka doll is artificial intelligence. Artificial intelligence, or AI, encompasses the ability of machines to perform intelligent and cognitive tasks. Comparable to the way the Industrial Revolution gave birth to an era of machines that could simulate physical tasks, AI is driving the development of machines capable of simulating cognitive abilities.

While still broad but dramatically more honed than computer science and data science, AI contains numerous subfields that are popular today. These subfields include search and planning, reasoning and knowledge representation, perception, natural language processing (NLP), and of course, machine learning. Machine learning bleeds into other fields of AI, including NLP and perception through the shared use of self-learning algorithms.

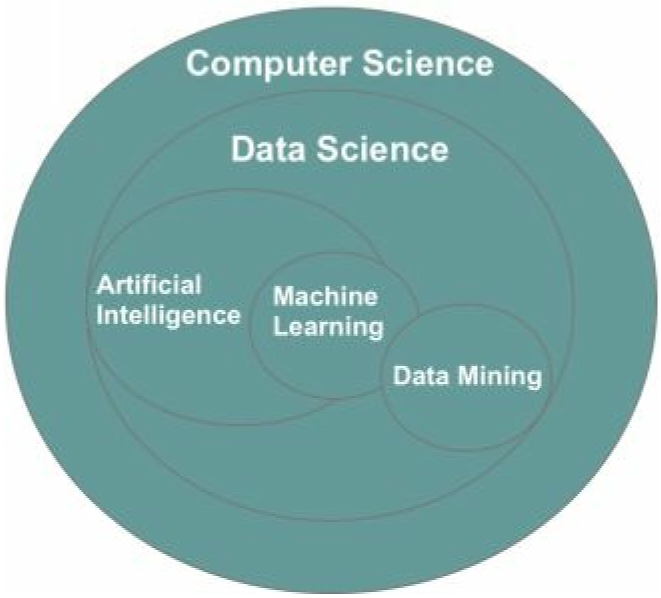


Figure 4: Visual representation of the relationship between data-related fields

For students with an interest in AI, machine learning provides an excellent starting point in that it offers a more narrow and practical lens of study compared to the conceptual ambiguity of AI. Algorithms found in machine learning can also be applied across other disciplines, including perception and natural language processing. In addition, a Master's degree is adequate to develop a certain level of expertise in machine learning, but you may need a PhD to make any true progress in AI.

As mentioned, machine learning also overlaps with data mining—a sister discipline that focuses on discovering and unearthing patterns in large datasets. Popular algorithms, such as *k*-means clustering, association analysis, and regression analysis, are applied in both data mining and machine learning to analyze data. But where machine learning focuses on the incremental process of self-learning and data modeling to form predictions about the future, data mining narrows in on cleaning large datasets to glean valuable insight from the past.

The difference between data mining and machine learning can be explained through an analogy of two teams of archaeologists. The first team is made up of archaeologists who focus their efforts on removing debris that lies in the way of valuable items, hiding them from direct sight. Their primary goals are to excavate the area, find new valuable discoveries, and then pack up their equipment and move on. A day later, they will fly to another exotic destination to start a new project with no relationship to the site they

excavated the day before.

The second team is also in the business of excavating historical sites, but these archaeologists use a different methodology. They deliberately reframe from excavating the main pit for several weeks. In that time, they visit other relevant archaeological sites in the area and examine how each site was excavated. After returning to the site of their own project, they apply this knowledge to excavate smaller pits surrounding the main pit.

The archaeologists then analyze the results. After reflecting on their experience excavating one pit, they optimize their efforts to excavate the next. This includes predicting the amount of time it takes to excavate a pit, understanding variance and patterns found in the local terrain and developing new strategies to reduce error and improve the accuracy of their work. From this experience, they are able to optimize their approach to form a strategic model to excavate the main pit.

If it is not already clear, the first team subscribes to data mining and the second team to machine learning. At a micro-level, both data mining and machine learning appear similar, and they do use many of the same tools. Both teams make a living excavating historical sites to discover valuable items. But in practice, their methodology is different. The machine learning team focuses on dividing their dataset into training data and test data to create a model, and improving future predictions based on previous experience. Meanwhile, the data mining team concentrates on excavating the target area as effectively as possible—without the use of a self-learning model—before moving on to the next cleanup job.

ML CATEGORIES

Machine learning incorporates several hundred statistical-based algorithms and choosing the right algorithm or combination of algorithms for the job is a constant challenge for anyone working in this field. But before we examine specific algorithms, it is important to understand the three overarching categories of machine learning. These three categories are **supervised**, **unsupervised,** and **reinforcement**.

Supervised Learning

As the first branch of machine learning, supervised learning concentrates on learning patterns through connecting the relationship between variables and known outcomes and working with labeled datasets.

Supervised learning works by feeding the machine sample data with various features (represented as “X”) and the correct value output of the data (represented as “y”). The fact that the output and feature values are known qualifies the dataset as “labeled.” The algorithm then deciphers patterns that exist in the data and creates a model that can reproduce the same underlying rules with new data.

For instance, to predict the market rate for the purchase of a used car, a supervised algorithm can formulate predictions by analyzing the relationship between car attributes (including the year of make, car brand, mileage, etc.) and the selling price of other cars sold based on historical data. Given that the supervised algorithm knows the final price of other cards sold, it can then work backward to determine the relationship between the characteristics of the car and its value.

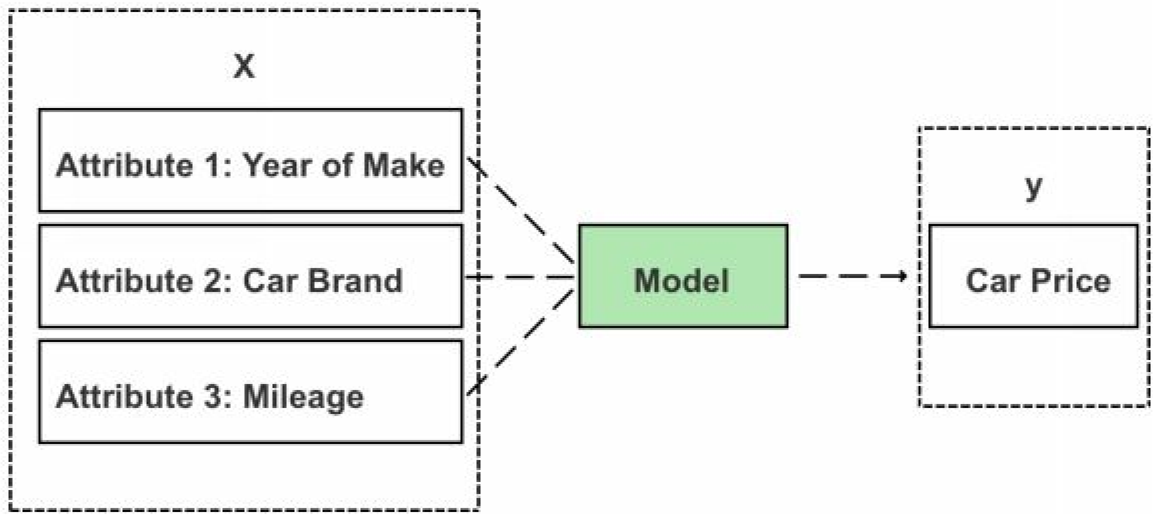


Figure 1: Car value prediction model

After the machine deciphers the rules and patterns of the data, it creates what is known as a model: an algorithmic equation for producing an outcome with new data based on the rules derived from the training data. Once the model is prepared, it can be applied to new data and tested for accuracy. After the model has passed both the training and test data stages, it is ready to be applied and used in the real world.

In Chapter 13, we will create a model for predicting house values where y is the actual house price and X are the variables that impact y, such as land size, location, and the number of rooms. Through supervised learning, we will create a rule to predict y (house value) based on the given values of various variables (X).

Examples of supervised learning algorithms include regression analysis, decision trees, *k*-nearest neighbors, neural networks, and support vector machines. Each of these techniques will be introduced later in the book.

Unsupervised Learning

In the case of unsupervised learning, not all variables and data patterns are classified. Instead, the machine must uncover hidden patterns and create labels through the use of unsupervised learning algorithms. The *k*-means clustering algorithm is a popular example of unsupervised learning. This simple algorithm groups data points that are found to possess similar features as shown in Figure 1.

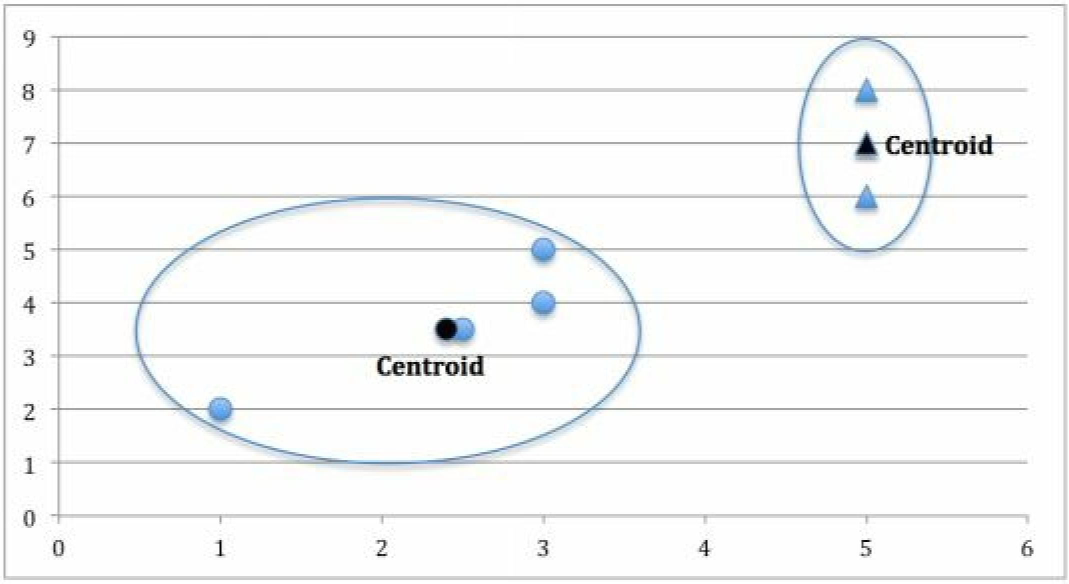


Figure 1: Example of *k*-means clustering, a popular unsupervised learning technique

If you group data points based on the purchasing behavior of SME (Small and Medium-sized Enterprises) and large enterprise customers, for example, you are likely to see two clusters emerge. This is because SMEs and large enterprises tend to have disparate buying habits. When it comes to purchasing cloud infrastructure, for instance, basic cloud hosting resources and a Content Delivery Network (CDN) may prove sufficient for most SME customers. Large enterprise customers, though, are more likely to purchase a wider array of cloud products and entire solutions that include advanced security and networking products like WAF (Web Application Firewall), a dedicated private connection, and VPC (Virtual Private Cloud). By analyzing customer purchasing habits, unsupervised learning is capable of identifying these two groups of customers without specific labels that classify the company as small, medium or large.

The advantage of unsupervised learning is it enables you to discover patterns in the data that you were unaware existed—such as the presence of two major customer types. Clustering techniques such as *k*-means clustering can also provide the springboard for conducting further analysis after discrete groups have been discovered.

In industry, unsupervised learning is particularly powerful in fraud detection—where the most dangerous attacks are often those yet to be classified. One real-world example is DataVisor, who essentially built their business model based on unsupervised learning.

Founded in 2013 in California, DataVisor protects customers from fraudulent

online activities, including spam, fake reviews, fake app installs, and fraudulent transactions. Whereas traditional fraud protection services draw on supervised learning models and rule engines, DataVisor uses unsupervised learning which enables them to detect unclassified categories of attacks in their early stages.

On their website, DataVisor explains that "to detect attacks, existing solutions rely on human experience to create rules or labeled training data to tune models. This means they are unable to detect new attacks that haven't already been identified by humans or labeled in training data."[5]

This means that traditional solutions analyze the chain of activity for a particular attack and then create rules to predict a repeat attack. Under this scenario, the dependent variable (y) is the event of an attack and the independent variables (X) are the common predictor variables of an attack. Examples of independent variables could be:

a) A sudden large order from an unknown user. I.E. established customers generally spend less than $100 per order, but a new user spends $8,000 in one order immediately upon registering their account.

b) A sudden surge of user ratings. I.E. As a typical author and bookseller on Amazon.com, it's uncommon for my first published work to receive more than one book review within the space of one to two days. In general, approximately 1 in 200 Amazon readers leave a book review and most books go weeks or months without a review. However, I commonly see competitors in this category (data science) attracting 20-50 reviews in one day! (Unsurprisingly, I also see Amazon removing these suspicious reviews weeks or months later.)

c) Identical or similar user reviews from different users. Following the same Amazon analogy, I often see user reviews of my book appear on other books several months later (sometimes with a reference to my name as the author still included in the review!). Again, Amazon eventually removes these fake reviews and suspends these accounts for breaking their terms of service.

d) Suspicious shipping address. I.E. For small businesses that routinely ship products to local customers, an order from a distant location (where they don't advertise their products) can in rare cases be an indicator of fraudulent or malicious activity.

Standalone activities such as a sudden large order or a distant shipping address may prove too little information to predict sophisticated

cybercriminal activity and more likely to lead to many false positives. But a model that monitors combinations of independent variables, such as a sudden large purchase order from the other side of the globe or a landslide of book reviews that reuse existing content will generally lead to more accurate predictions. A supervised learning-based model could deconstruct and classify what these common independent variables are and design a detection system to identify and prevent repeat offenses.

Sophisticated cybercriminals, though, learn to evade classification-based rule engines by modifying their tactics. In addition, leading up to an attack, attackers often register and operate single or multiple accounts and incubate these accounts with activities that mimic legitimate users. They then utilize their established account history to evade detection systems, which are trigger-heavy against recently registered accounts. Supervised learning-based solutions struggle to detect sleeper cells until the actual damage has been made and especially with regard to new categories of attacks.

DataVisor and other anti-fraud solution providers therefore leverage unsupervised learning to address the limitations of supervised learning by analyzing patterns across hundreds of millions of accounts and identifying suspicious connections between users—without knowing the actual category of future attacks. By grouping malicious actors and analyzing their connections to other accounts, they are able to prevent new types of attacks whose independent variables are still unlabeled and unclassified. Sleeper cells in their incubation stage (mimicking legitimate users) are also identified through their association to malicious accounts. Clustering algorithms such as *k*-means clustering can generate these groupings without a full training dataset in the form of independent variables that clearly label indications of an attack, such as the four examples listed earlier. Knowledge of the dependent variable (known attackers) is generally the key to identifying other attackers before the next attack occurs. The other plus side of unsupervised learning is companies like DataVisor can uncover entire criminal rings by identifying subtle correlations across users.

We will cover unsupervised learning later in this book specific to clustering analysis. Other examples of unsupervised learning include association analysis, social network analysis, and descending dimension algorithms.

Reinforcement Learning

Reinforcement learning is the third and most advanced algorithm category in

machine learning. Unlike supervised and unsupervised learning, reinforcement learning continuously improves its model by leveraging feedback from previous iterations. This is different to supervised and unsupervised learning, which both reach an indefinite endpoint after a model is formulated from the training and test data segments.

Reinforcement learning can be complicated and is probably best explained through an analogy to a video game. As a player progresses through the virtual space of a game, they learn the value of various actions under different conditions and become more familiar with the field of play. Those learned values then inform and influence a player's subsequent behavior and their performance immediately improves based on their learning and past experience.

Reinforcement learning is very similar, where algorithms are set to train the model through continuous learning. A standard reinforcement learning model has measurable performance criteria where outputs are not tagged—instead, they are graded. In the case of self-driving vehicles, avoiding a crash will allocate a positive score and in the case of chess, avoiding defeat will likewise receive a positive score.

A specific algorithmic example of reinforcement learning is Q-learning. In Q-learning, you start with a set environment of *states,* represented by the symbol 'S'. In the game Pac-Man, states could be the challenges, obstacles or pathways that exist in the game. There may exist a wall to the left, a ghost to the right, and a power pill above—each representing different *states*.

The set of possible actions to respond to these states is referred to as "A." In the case of Pac-Man, actions are limited to left, right, up, and down movements, as well as multiple combinations thereof.

The third important symbol is "Q." Q is the starting value and has an initial value of "0."

As Pac-Man explores the space inside the game, two main things will happen:

1) Q drops as negative things occur after a given state/action

2) Q increases as positive things occur after a given state/action

In Q-learning, the machine will learn to match the action for a given state that generates or maintains the highest level of Q. It will learn initially through the process of random movements (actions) under different conditions (states). The machine will record its results (rewards and penalties) and how they impact its Q level and store those values to inform and optimize its future

actions.

While this sounds simple enough, implementation is a much more difficult task and beyond the scope of an absolute beginner's introduction to machine learning. Reinforcement learning algorithms aren't covered in this book, however, I will leave you with a link to a more comprehensive explanation of reinforcement learning and Q-learning following the Pac-Man scenario.

https://inst.eecs.berkeley.edu/~cs188/sp12/projects/reinforcement/reinforcement.html

THE ML TOOLBOX

A handy way to learn a new subject area is to map and visualize the essential materials and tools inside a toolbox.

If you were packing a toolbox to build websites, for example, you would first pack a selection of programming languages. This would include frontend languages such as HTML, CSS, and JavaScript, one or two backend programming languages based on personal preferences, and of course, a text editor. You might throw in a website builder such as WordPress and then have another compartment filled with web hosting, DNS, and maybe a few domain names that you've recently purchased.

This is not an extensive inventory, but from this general list, you can start to gain a better appreciation of what tools you need to master in order to become a successful website developer.

Let's now unpack the toolbox for machine learning.

Compartment 1: Data

In the first compartment is your data. Data constitutes the input variables needed to form a prediction. Data comes in many forms, including structured and non-structured data. As a beginner, it is recommended that you start with *structured data*. This means that the data is defined and labeled (with schema) in a table, as shown here:

Date	Bitcoin Price	No. of Days Transpired
19-05-2015	234.31	1
14-01-2016	431.76	240
09-07-2016	652.14	417
15-01-2017	817.26	607
24-05-2017	2358.96	736

Before we proceed, I first want to explain the anatomy of a tabular dataset. A tabular (table-based) dataset contains data organized in rows and columns. In each column is a *feature*. A feature is also known as a *variable,* a *dimension* or an *attribute*—but they all mean the same thing.

Each individual row represents a single observation of a given feature/variable. Rows are sometimes referred to as a *case* or *value*, but in this book, we will use the term "row."

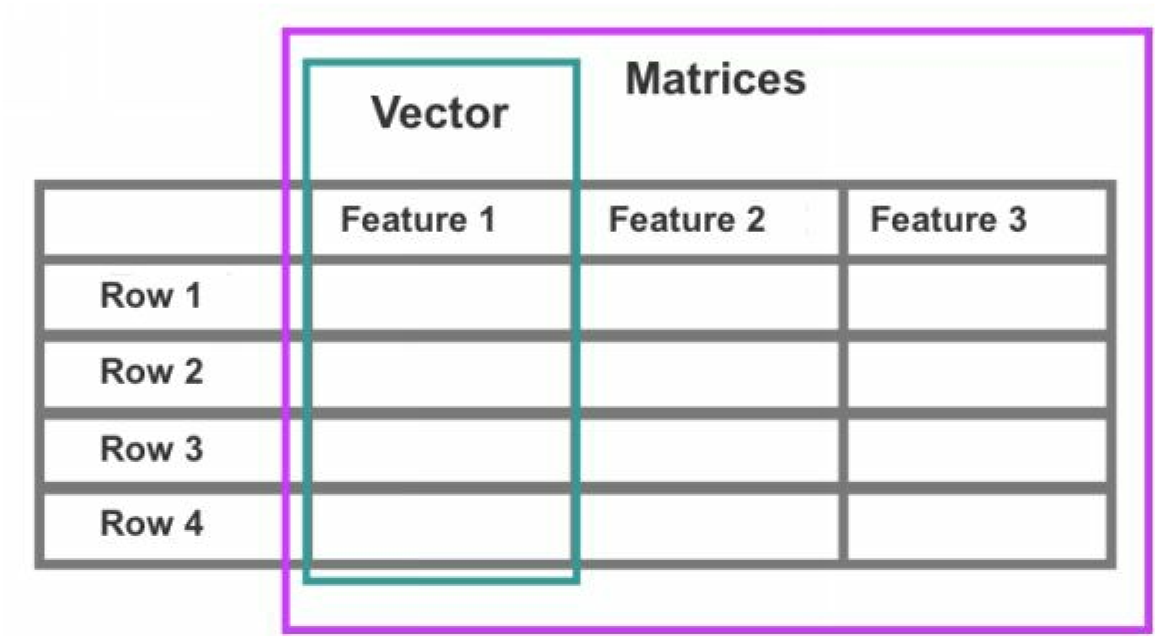


Figure 1: Example of a tabular dataset

Each column is known as a *vector*. Vectors store your X and y values and multiple vectors (columns) are commonly referred to as *matrices*. In the case of supervised learning, y will already exist in your dataset and be used to identify patterns in relation to independent variables (X). The y values are commonly expressed in the final column, as shown in Figure 2.

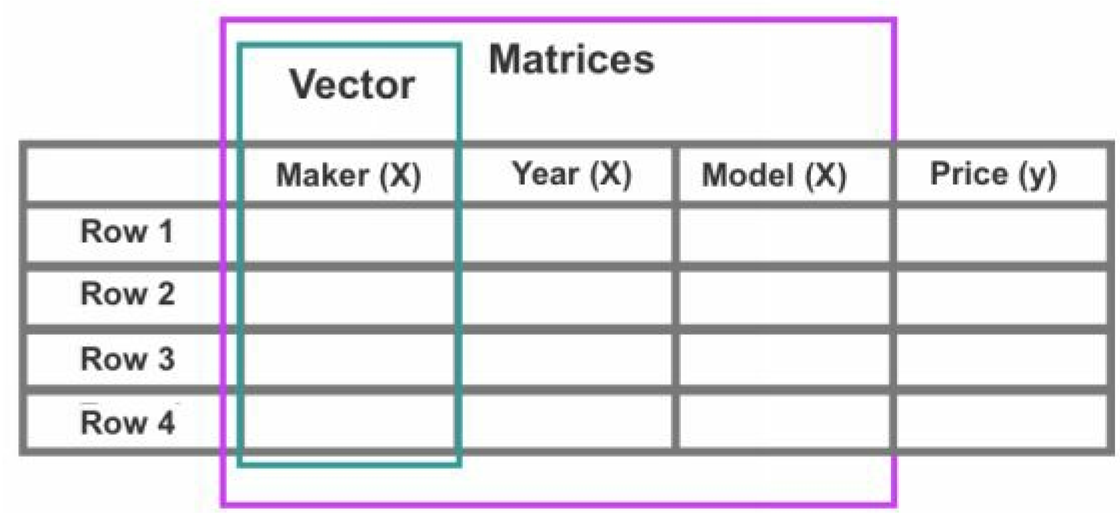
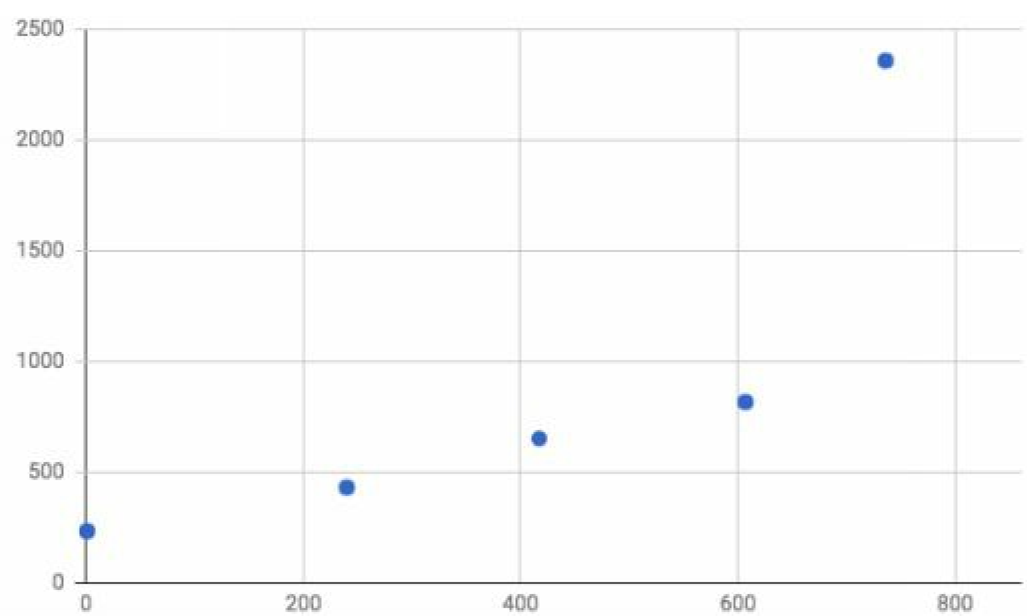


Figure 2: The y value is often but not always expressed in the far right column

Next, within the first compartment of the toolbox is a range of scatterplots, including 2-D, 3-D, and 4-D plots. A 2-D scatterplot consists of a vertical axis (known as the y-axis) and a horizontal axis (known as the x-axis) and provides the graphical canvas to plot a series of dots, known as data points. Each data point on the scatterplot represents one observation from the dataset, with X values plotted on the x-axis and y values plotted on the y-axis.



	Independent Variable (X)	Dependent Variable (y)
Row 1	1	243.31
Row 2	240	431.76
Row 3	417	653.14
Row 4	607	817.26
Row 5	736	2358.96

Figure 3: Example of a 2-D scatterplot. X represents days passed since the recording of Bitcoin prices and y represents recorded Bitcoin price.

Compartment 2: Infrastructure

The second compartment of the toolbox contains your infrastructure, which consists of platforms and tools to process data. As a beginner to machine learning, you are likely to be using a web application (such as Jupyter Notebook) and a programming language like Python. There are then a series of machine learning libraries, including NumPy, Pandas, and Scikit-learn that are compatible with Python. Machine learning libraries are a collection of pre-compiled programming routines frequently used in machine learning.

You will also need a machine from which to work, in the form of a computer or a virtual server. In addition, you may need specialized libraries for data visualization such as Seaborn and Matplotlib, or a standalone software program like Tableau, which supports a range of visualization techniques including charts, graphs, maps, and other visual options.

With your infrastructure sprayed out across the table (hypothetically of course), you are now ready to get to work building your first machine learning model. The first step is to crank up your computer. Laptops and desktop computers are both suitable for working with smaller datasets. You will then need to install a programming environment, such as Jupyter Notebook, and a programming language, which for most beginners is Python. Python is the most widely used programming language for machine learning because:

a) It is easy to learn and operate,
b) It is compatible with a range of machine learning libraries, and
c) It can be used for related tasks, including data collection (web scraping) and data piping (Hadoop and Spark).

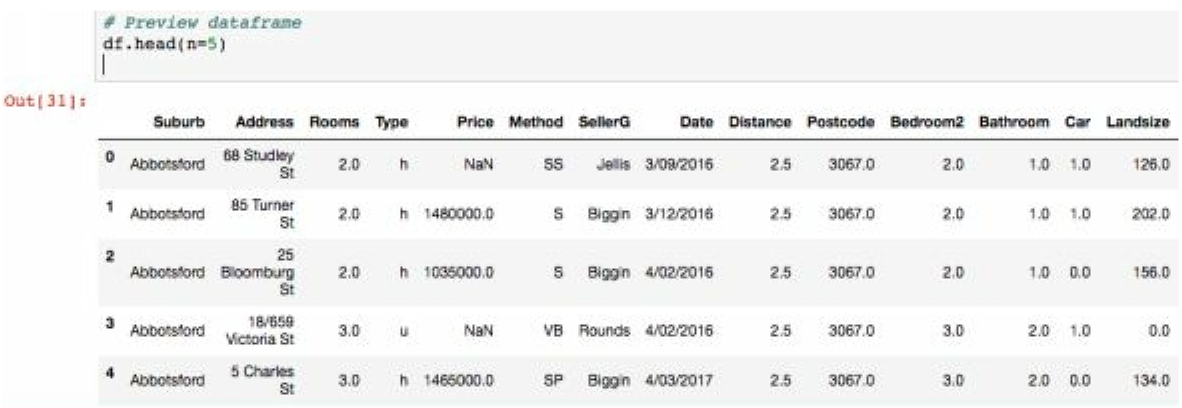
Other go-to languages for machine learning include C and C++. If you're proficient with C and C++ then it makes sense to stick with what you already

know. C and C++ are the default programming languages for advanced machine learning because they can run directly on a GPU (Graphical Processing Unit). Python needs to be converted first before it can run on a GPU, but we will get to this and what a GPU is later in the chapter.

Next, Python users will typically install the following libraries: NumPy, Pandas, and Scikit-learn. NumPy is a free and open-source library that allows you to efficiently load and work with large datasets, including managing matrices.

Scikit-learn provides access to a range of popular algorithms, including linear regression, Bayes' classifier, and support vector machines.

Finally, Pandas enables your data to be represented on a virtual spreadsheet that you can control through code. It shares many of the same features as Microsoft Excel in that it allows you to edit data and perform calculations. In fact, the name Pandas derives from the term "panel data," which refers to its ability to create a series of panels, similar to "sheets" in Excel. Pandas is also ideal for importing and extracting data from CSV files.



```
# Preview dataframe
df.head(n=5)
```

Out[31]:

	Suburb	Address	Rooms	Type	Price	Method	SellerG	Date	Distance	Postcode	Bedroom2	Bathroom	Car	Landsize
0	Abbotsford	68 Studley St	2.0	h	NaN	SS	Jellis	3/09/2016	2.5	3067.0	2.0	1.0	1.0	126.0
1	Abbotsford	85 Turner St	2.0	h	1480000.0	S	Biggin	3/12/2016	2.5	3067.0	2.0	1.0	1.0	202.0
2	Abbotsford	25 Bloomburg St	2.0	h	1035000.0	S	Biggin	4/02/2016	2.5	3067.0	2.0	1.0	0.0	156.0
3	Abbotsford	18/659 Victoria St	3.0	u	NaN	VB	Rounds	4/02/2016	2.5	3067.0	3.0	2.0	1.0	0.0
4	Abbotsford	5 Charles St	3.0	h	1465000.0	SP	Biggin	4/03/2017	2.5	3067.0	3.0	2.0	0.0	134.0

Figure 4: Previewing a table in Jupyter Notebook using Pandas

In summary, users can draw on these three libraries to:

1) Load and work with a dataset via NumPy.

2) Clean up and perform calculations on data, and extract data from CSV files with Pandas.

3) Implement algorithms with Scikit-learn.

For students seeking alternative programming options (beyond Python, C, and C++), other relevant programming languages for machine learning include R, MATLAB, and Octave.

R is a free and open-source programming language optimized for

mathematical operations, and conducive to building matrices and statistical functions, which are built directly into the language libraries of R. Although R is commonly used for data analytics and data mining, R supports machine learning operations as well.

MATLAB and Octave are direct competitors to R. MATLAB is a commercial and propriety programming language. It is strong in regards to solving algebraic equations and is also a quick programming language to learn. MATLAB is widely used in electrical engineering, chemical engineering, civil engineering, and aeronautical engineering. However, computer scientists and computer engineers tend not to rely on MATLAB as heavily and especially in recent times. In machine learning, MATLAB is more often used in academia than in industry. Thus, while you may see MATLAB featured in online courses, and especially on Coursera, this is not to say that it's commonly used in the wild. If, however, you're coming from an engineering background, MATLAB is certainly a logical choice.

Lastly, Octave is essentially a free version of MATLAB developed in response to MATLAB by the open-source community.

Compartment 3: Algorithms

Now that the machine learning environment is set up and you've chosen your programming language and libraries, you can next import your data directly from a CSV file. You can find hundreds of interesting datasets in CSV format from kaggle.com. After registering as a member of their platform, you can download a dataset of your choice. Best of all, Kaggle datasets are free and there is no cost to register as a user.

The dataset will download directly to your computer as a CSV file, which means you can use Microsoft Excel to open and even perform basic algorithms such as linear regression on your dataset.

Next is the third and final compartment that stores the algorithms. Beginners will typically start off by using simple supervised learning algorithms such as linear regression, logistic regression, decision trees, and *k*-nearest neighbors. Beginners are also likely to apply unsupervised learning in the form of *k*-means clustering and descending dimension algorithms.

Visualization

No matter how impactful and insightful your data discoveries are, you need a

way to effectively communicate the results to relevant decision-makers. This is where data visualization, a highly effective medium to communicate data findings to a general audience, comes in handy. The visual message conveyed through graphs, scatterplots, box plots, and the representation of numbers in shapes makes for quick and easy storytelling.

In general, the less informed your audience is, the more important it is to visualize your findings. Conversely, if your audience is knowledgeable about the topic, additional details and technical terms can be used to supplement visual elements.

To visualize your results you can draw on Tableau or a Python library such as Seaborn, which are stored in the second compartment of the toolbox.

Advanced Toolbox

We have so far examined the toolbox for a typical beginner, but what about an advanced user? What would their toolbox look like? While it may take some time before you get to work with the advanced toolkit, it doesn't hurt to have a sneak peek.

The toolbox for an advanced learner resembles the beginner's toolbox but naturally comes with a broader spectrum of tools and, of course, data. One of the biggest differences between a beginner and an advanced learner is the size of the data they manage and operate. Beginners naturally start by working with small datasets that are easy to manage and which can be downloaded directly to one's desktop as a simple CSV file. Advanced learners, though, will be eager to tackle massive datasets, well in the vicinity of big data.

Compartment 1: Big Data

Big data is used to describe a dataset that, due to its value, variety, volume, and velocity, defies conventional methods of processing and would be impossible for a human to process without the assistance of an advanced machine. Big data does not have an exact definition in terms of size or the total number of rows and columns. At the moment, petabytes qualify as big data, but datasets are becoming increasingly larger as we find new ways to efficiently collect and store data at low cost. And with big data also comes greater noise and complicated data structures. A huge part, therefore, of working with big data is *scrubbing*: the process of refining your dataset before building your model, which will be covered in the next chapter.

Compartment 2: Infrastructure

After scrubbing the dataset, the next step is to pull out your machine learning equipment. In terms of tools, there are no real surprises. Advanced learners are still using the same machine learning libraries, programming languages, and programming environments as beginners.

However, given that advanced learners are now dealing with up to petabytes of data, robust infrastructure is required. Instead of relying on the CPU of a personal computer, advanced students typically turn to distributed computing and a cloud provider such as Amazon Web Services (AWS) to run their data processing on what is known as a Graphical Processing Unit (GPU) instance.

GPU chips were originally added to PC motherboards and video consoles such as the PlayStation 2 and the Xbox for gaming purposes. They were developed to accelerate the creation of images with millions of pixels whose frames needed to be constantly recalculated to display output in less than a second. By 2005, GPU chips were produced in such large quantities that their price had dropped dramatically and they'd essentially matured into a commodity. Although highly popular in the video game industry, the application of such computer chips in the space of machine learning was not fully understood or realized until recently.

In his 2016 novel, *The Inevitable: Understanding the 12 Technological Forces That Will Shape Our Future*, Founding Executive Editor of Wired Magazine, Kevin Kelly, explains that in 2009, Andrew Ng and a team at Stanford University discovered how to link inexpensive GPU clusters to run neural networks consisting of hundreds of millions of node connections.

"Traditional processors required several weeks to calculate all the cascading possibilities in a neural net with one hundred million parameters. Ng found that a cluster of GPUs could accomplish the same thing in a day."[6]

As a specialized parallel computing chip, GPU instances are able to perform many more floating point operations per second than a CPU, allowing for much faster solutions with linear algebra and statistics than with a CPU.

It is important to note that C and C++ are the preferred languages to directly edit and perform mathematical operations on the GPU. However, Python can also be used and converted into C in combination with TensorFlow from Google.

Although it's possible to run TensorFlow on the CPU, you can gain up to about 1,000x in performance using the GPU. Unfortunately for Mac users, TensorFlow is only compatible with the Nvidia GPU card, which is no longer available with Mac OS X. Mac users can still run TensorFlow on their CPU but will need to engineer a patch/external driver or run their workload on the cloud to access GPU. Amazon Web Services, Microsoft Azure, Alibaba Cloud, Google Cloud Platform, and other cloud providers offer pay-as-you-go GPU resources, which may start off free through a free trial program. Google Cloud Platform is currently regarded as a leading option for GPU resources based on performance and pricing. In 2016, Google also announced that it would publicly release a Tensor Processing Unit designed specifically for running TensorFlow, which is already used internally at Google.

Compartment 3: Advanced Algorithms

To round out this chapter, let’s have a look at the third compartment of the advanced toolbox containing machine learning algorithms.

To analyze large datasets, advanced learners work with a plethora of advanced algorithms including Markov models, support vector machines, and Q-learning, as well as a series of simple algorithms like those found in the beginner’s toolbox. But the algorithm family they’re most likely to use is neural networks (introduced in Chapter 10), which comes with its own selection of advanced machine learning libraries.

While Scikit-learn offers a range of popular shallow algorithms, TensorFlow is the machine learning library of choice for deep learning/neural networks as it supports numerous advanced techniques including automatic calculus for back-propagation/gradient descent. Due to the depth of resources, documentation, and jobs available with TensorFlow, it is the obvious framework to learn today.

Popular alternative neural network libraries include Torch, Caffe, and the fast-growing Keras. Written in Python, Keras is an open-source deep learning library that runs on top of TensorFlow, Theano, and other frameworks, and allows users to perform fast experimentation in fewer lines of code. Like a WordPress website theme, Keras is minimal, modular, and quick to get up and running but is less flexible compared with TensorFlow and other libraries. Users will sometimes utilize Keras to validate their model before switching to TensorFlow to build a more customized model.

Caffe is also open-source and commonly used to develop deep learning architectures for image classification and image segmentation. Caffe is written in C++ but has a Python interface that also supports GPU-based acceleration using the Nvidia CuDNN.

Released in 2002, Torch is well established in the deep learning community. It is open-source and based on the programming language Lua. Torch offers a range of algorithms for deep learning and is used within Facebook, Google, Twitter, NYU, IDIAP, Purdue as well as other companies and research labs.[7]

Until recently, Theano was another competitor to TensorFlow but as of late 2017, contributions to the framework have officially ceased.

Sometimes used beside neural networks is another advanced approach called ensemble modeling. This technique essentially combines algorithms and statistical techniques to create a unified model, which we will explore further in Chapter 12.

DATA SCRUBBING

Much like many categories of fruit, datasets nearly always require some form of upfront cleaning and human manipulation before they are ready to digest. For machine learning and data science more broadly, there are a vast number of techniques to scrub data.

Scrubbing is the technical process of refining your dataset to make it more workable. This can involve modifying and sometimes removing incomplete, incorrectly formatted, irrelevant or duplicated data. It can also entail converting text-based data to numerical values and the redesigning of features. For data practitioners, data scrubbing usually demands the greatest application of time and effort.

Feature Selection

To generate the best results from your data, it is important to first identify the variables most relevant to your hypothesis. In practice, this means being selective about the variables you select to design your model.

Rather than creating a four-dimensional scatterplot with four features in the model, an opportunity may present to select two highly relevant features and build a two-dimensional plot that is easier to interpret. Moreover, preserving features that do not correlate strongly with the outcome value can, in fact, manipulate and derail the model’s accuracy. Consider the following table excerpt downloaded from kaggle.com documenting dying languages.

Name in English	Name in Spanish	Countries	Country Code
South Italian	Napolitano-calabres	Italy	ITA
Sicilian	Siciliano	Italy	ITA
Low Saxon	Bajo Sajón	Germany, Denmark, Netherlands, Poland, Russian Federation	DEU, DNK, NLD, POL, RUS
Belarusian	Bielorruso	Belarus, Latvia, Lithuania, Poland, Russian Federation, Ukraine	BRB, LVA, LTU, POL, RUS, UKR
Lombard	Lombardo	Italy, Switzerland	ITA, CHE
Romani	Romaní	Albania, Germany, Austria, Belarus, Bosnia and Herzegovina, Bulgaria, Croatia, Estonia, Finland, France, Greece, Hungary, Italy, Latvia, Lithuania, The former Yugoslav Republic of Macedonia, Netherlands, Poland, Romania, United Kingdom of Great Britain and Northern Ireland, Russian Federation, Slovakia, Slovenia, Switzerland, Czech Republic, Turkey, Ukraine, Serbia, Montenegro	ALB, DEU, AUT, BRB, BIH, BGR, HRV, EST, FIN, FRA, GRC, HUN, ITA, LVA, LTU, MKD, NLD, POL, ROU, GBR, RUS, SVK, SVN, CHE, CZE, TUR, UKR, SRB, MNE
Yiddish	Yiddish	Israel	ISR
Gondi	Gondi	India	IND

Database: https://www.kaggle.com/the-guardian/extinct-languages

Let's say our goal is to identify variables that lead to a language becoming endangered. Based on this goal, it's unlikely that a language's "Name in Spanish" will lead to any relevant insight. We can therefore go ahead and delete this vector (column) from the dataset. This will help to prevent over-complication and potential inaccuracies, and will also improve the overall processing speed of the model.

Secondly, the dataset holds duplicate information in the form of separate vectors for "Countries" and "Country Code." Including both of these vectors doesn't provide any additional insight; hence, we can choose to delete one

and retain the other.

Another method to reduce the number of features is to roll multiple features into one. In the next table, we have a list of products sold on an e-commerce platform. The dataset comprises four buyers and eight products. This is not a large sample size of buyers and products—due in part to the spatial limitations of the book format. A real-life e-commerce platform would have many more columns to work with, but let's go ahead with this example.

	Protein Shake	Nike Sneakers	Adidas Boots	Fitbit	Powerade	Protein Bar	Fitness Watch	Vitamins
Buyer 1	1	1	0	1	0	5	1	0
Buyer 2	0	0	0	0	0	0	0	1
Buyer 3	3	0	1	0	5	0	0	0
Buyer 4	1	1	0	0	10	1	0	0

In order to analyze the data in a more efficient way, we can reduce the number of columns by merging similar features into fewer columns. For instance, we can remove individual product names and replace the eight product items with a lower number of categories or subtypes. As all product items fall under the single category of "fitness," we will sort by product subtype and compress the columns from eight to three. The three newly created product subtype columns are "Health Food," "Apparel," and "Digital."

	Health Food	Apparel	Digital
Buyer 1	6	1	2
Buyer 2	1	0	0
Buyer 3	8	1	0
Buyer 4	12	1	0

This enables us to transform the dataset in a way that preserves and captures information using fewer variables. The downside to this transformation is that we have less information about relationships between specific products.

Rather than recommending products to users according to other individual products, recommendations will instead be based on relationships between product subtypes.

Nonetheless, this approach does uphold a high level of data relevancy. Buyers will be recommended health food when they buy other health food or when they buy apparel (depending on the level of correlation), and obviously not machine learning textbooks—unless it turns out that there is a strong correlation there! But alas, such a variable is outside the frame of this dataset.

Remember that data reduction is also a business decision, and business owners in counsel with the data science team will need to consider the trade-off between convenience and the overall precision of the model.

Row Compression

In addition to feature selection, there may also be an opportunity to reduce the number of rows and thereby compress the total number of data points. This can involve merging two or more rows into one. For example, in the following dataset, "Tiger" and "Lion" can be merged and renamed "Carnivore."

Before

Animal	Meat Eater	Legs	Tail	Race Time
Tiger	Yes	4	Yes	2:01 mins
Lion	Yes	4	Yes	2:05 mins
Tortoise	No	4	No	55:02 mins

After

Animal	Meat Eater	Legs	Tail	Race Time
Carnivore	Yes	4	Yes	2:03 mins
Tortoise	No	4	No	55:02 mins

However, by merging these two rows (Tiger & Lion), the feature values for

both rows must also be aggregated and recorded in a single row. In this case, it is viable to merge the two rows because they both possess the same categorical values for all features except y (Race Time)—which can be aggregated. The race time of the Tiger and the Lion can be added and divided by two.

Numerical values, such as time, are normally simple to aggregate unless they are categorical. For instance, it would be impossible to aggregate an animal with four legs and an animal with two legs! We obviously can't merge these two animals and set "three" as the aggregate number of legs.

Row compression can also be difficult to implement when numerical values aren't available. For example, the values "Japan" and "Argentina" are very difficult to merge. The countries "Japan" and "South Korea" can be merged, as they can be categorized as the same continent, "Asia" or "East Asia." However, if we add "Pakistan" and "Indonesia" to the same group, we may begin to see skewed results, as there are significant cultural, religious, economic, and other dissimilarities between these four countries.

In summary, non-numerical and categorical row values can be problematic to merge while preserving the true value of the original data. Also, row compression is normally less attainable than feature compression for most datasets.

One-hot Encoding

After choosing variables and rows, you next want to look for text-based features that can be converted into numbers. Aside from set text-based values such as True/False (that automatically convert to "1" and "0" respectively), many algorithms and also scatterplots are not compatible with non-numerical data.

One means to convert text-based features into numerical values is through *one-hot encoding*, which transforms features into binary form, represented as "1" or "0"—"True" or "False." A "0," representing False, means that the feature does not belong to a particular category, whereas a "1"—True or "hot"—denotes that the feature does belong to a set category.

Below is another excerpt of the dataset on dying languages, which we can use to practice one-hot encoding.

Name in English	Speakers	Degree of Endangerment
South Italian	7500000	Vulnerable
Sicilian	5000000	Vulnerable
Low Saxon	4800000	Vulnerable
Belarusian	4000000	Vulnerable
Lombard	3500000	Definitely endangered
Romani	3500000	Definitely endangered
Yiddish	3000000	Definitely endangered
Gondi	2713790	Vulnerable
Picard	700000	Severely endangered

First, note that the values contained in the "No. of Speakers" column do not contain commas or spaces, e.g. 7,500,000 and 7 500 000. Although such formatting does make large numbers clearer for our eyes, programming languages don't require such niceties. In fact, formatting numbers can lead to an invalid syntax or trigger an unwanted result, depending on the programming language you use. So remember to keep numbers unformatted for programming purposes. Feel free, though, to add spacing or commas at the data visualization stage, as this will make it easier for your audience to interpret!

On the right-hand-side of the table is a vector categorizing the degree of endangerment of the nine different languages. This column we can convert to numerical values by applying the one-hot encoding method, as demonstrated in the subsequent table.

Name in English	Speakers	Vulnerable	Definitely Endangered	Severely Endangered
South Italian	7500000	1	0	0
Sicilian	5000000	1	0	0
Low Saxon	4800000	1	0	0
Belarusian	4000000	1	0	0
Lombard	3500000	0	1	0
Romani	3500000	0	1	0
Yiddish	3000000	0	1	0
Gondi	2713790	1	0	0
Picard	700000	0	0	1

Using one-hot encoding, the dataset has expanded to five columns and we have created three new features from the original feature (Degree of Endangerment). We have also set each column value to "1" or "0," depending on the original category value.

This now makes it possible for us to input the data into our model and choose from a wider array of machine learning algorithms. The downside is that we have more dataset features, which may lead to slightly longer processing time. This is nonetheless manageable, but it can be problematic for datasets where original features are split into a larger number of new features.

One hack to minimize the number of features is to restrict binary cases to a single column. As an example, there is a speed dating dataset on kaggle.com that lists "Gender" in a single column using one-hot encoding. Rather than create discrete columns for both "Male" and "Female," they merged these two features into one. According to the dataset's key, females are denoted as "0" and males are denoted as "1." The creator of the dataset also used this technique for "Same Race" and "Match."

Subject Number ID	Gender	Same Race	Age	Match
1	0	0	27	0
1	0	0	22	0
1	0	1	22	1
1	0	0	23	1
1	0	0	24	1
1	0	0	25	0
1	0	0	30	0

Gender:	Same Race:	Match:
Female = 0	No = 0	No = 0
Male = 1	Yes = 1	Yes = 1

Database: https://www.kaggle.com/annavictoria/speed-dating-experiment

Binning

Binning is another method of feature engineering that is used to convert numerical values into a category.

Whoa, hold on! Didn't you say that numerical values were a good thing? Yes, numerical values tend to be preferred in most cases. Where numerical values are less ideal, is in situations where they list variations irrelevant to the goals of your analysis. Let's take house price evaluation as an example. The exact measurements of a tennis court might not matter greatly when evaluating house prices. The relevant information is whether the house *has* a tennis court. The same logic probably also applies to the garage and the swimming pool, where the existence or non-existence of the variable is more influential than their specific measurements.

The solution here is to replace the numeric measurements of the tennis court with a True/False feature or a categorical value such as "small," "medium," and "large." Another alternative would be to apply one-hot encoding with "0" for homes that *do not* have a tennis court and "1" for homes that *do* have a

tennis court.

Missing Data

Dealing with missing data is never a desired situation. Imagine unpacking a jigsaw puzzle that you discover has five percent of its pieces missing. Missing values in a dataset can be equally frustrating and will ultimately interfere with your analysis and final predictions. There are, however, strategies to minimize the negative impact of missing data.

One approach is to approximate missing values using the *mode* value. The mode represents the single most common variable value available in the dataset. This works best with categorical and binary variable types.

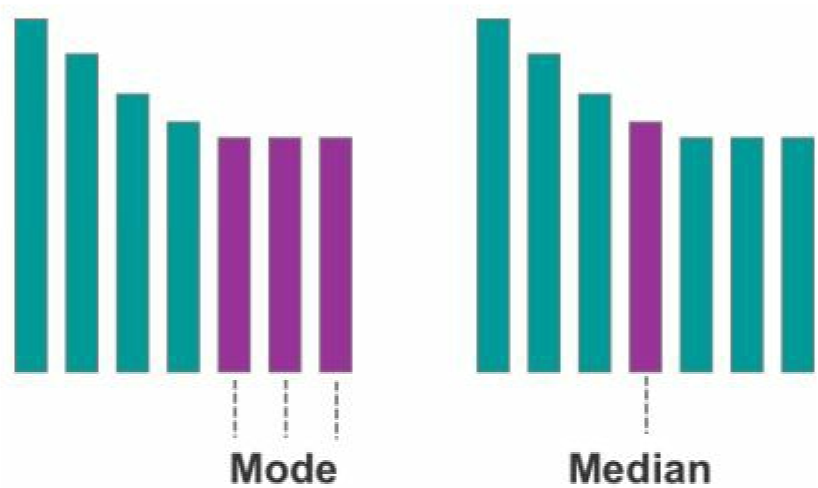


Figure 1: A visual example of the mode and median respectively

The second approach to manage missing data is to approximate missing values using the *median* value, which adopts the value(s) located in the middle of the dataset. This works best with integers (whole numbers) and continuous variables (numbers with decimals).

As a last resort, rows with missing values can be removed altogether. The obvious downside to this approach is having less data to analyze and potentially less comprehensive results.

SETTING UP YOUR DATA

Once you have cleaned your dataset, the next job is to split the data into two segments for testing and training. It is very important not to test your model with the same data that you used for training. The ratio of the two splits should be approximately 70/30 or 80/20. This means that your training data should account for 70 percent to 80 percent of the rows in your dataset, and the other 20 percent to 30 percent of rows is your test data. It is vital to split your data by rows and not columns.

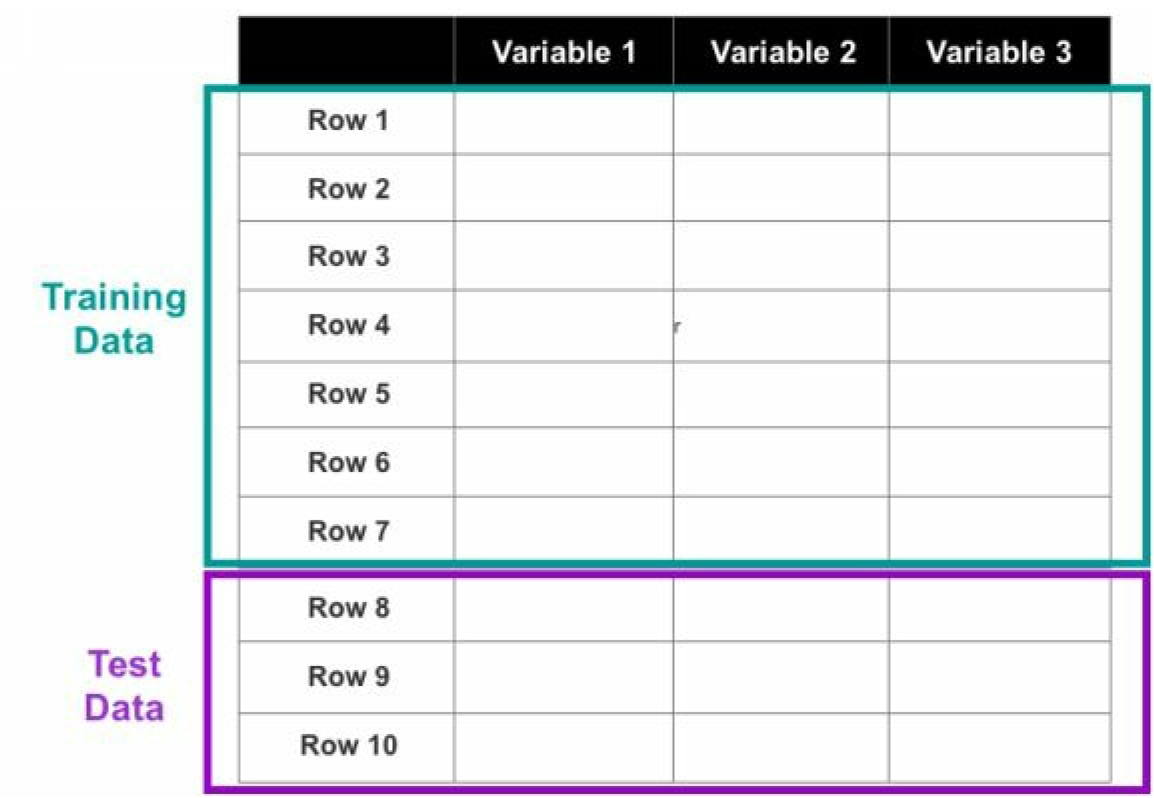


Figure 1: Training and test partitioning of the dataset 70/30

Before you split your data, it is important that you randomize all rows in the dataset. This helps to avoid bias in your model, as your original dataset might be arranged sequentially depending on the time it was collected or some other factor. Unless you randomize your data, you may accidentally omit important variance from the training data that will cause unwanted surprises when you

apply the trained model to your test data. Fortunately, Scikit-learn provides a built-in function to shuffle and randomize your data with just one line of code (demonstrated in Chapter 13).

After randomizing your data, you can begin to design your model and apply that to the training data. The remaining 30 percent or so of data is put to the side and reserved for testing the accuracy of the model.

In the case of supervised learning, the model is developed by feeding the machine the training data and the expected output (y). The machine is able to analyze and discern relationships between the features (X) found in the training data to calculate the final output (y).

The next step is to measure how well the model actually performs. A common approach to analyzing prediction accuracy is a measure called *mean absolute error*, which examines each prediction in the model and provides an average error score for each prediction.

In Scikit-learn, mean absolute error is found using the model.predict function on X (features). This works by first plugging in the y values from the training dataset and generating a prediction for each row in the dataset. Scikit-learn will compare the predictions of the model to the correct outcome and measure its accuracy. You will know if your model is accurate when the error rate between the training and test dataset is low. This means that the model has learned the dataset's underlying patterns and trends.

Once the model can adequately predict the values of the test data, it is ready for use in the wild. If the model fails to accurately predict values from the test data, you will need to check whether the training and test data were properly randomized. Alternatively, you may need to change the model's hyperparameters.

Each algorithm has hyperparameters; these are your algorithm settings. In simple terms, these settings control and impact how fast the model learns patterns and which patterns to identify and analyze.

Cross Validation

Although the training/test data split can be effective in developing models from existing data, a question mark remains as to whether the model will work on new data. If your existing dataset is too small to construct an accurate model, or if the training/test partition of data is not appropriate, this can lead to poor estimations of performance in the wild.

Fortunately, there is an effective workaround for this issue. Rather than splitting the data into two segments (one for training and one for testing), we can implement what is known as *cross validation*. Cross validation maximizes the availability of training data by splitting data into various combinations and testing each specific combination.

Cross validation can be performed through two primary methods. The first method is *exhaustive cross validation*, which involves finding and testing all possible combinations to divide the original sample into a training set and a test set. The alternative and more common method is non-exhaustive cross validation, known as *k-fold validation*. The *k*-fold validation technique involves splitting data into *k* assigned buckets and reserving one of those buckets to test the training model at each round.

To perform *k*-fold validation, data are first randomly assigned to *k* number of equal sized buckets. One bucket is then reserved as the test bucket and is used to measure and evaluate the performance of the remaining (*k*-1) buckets.

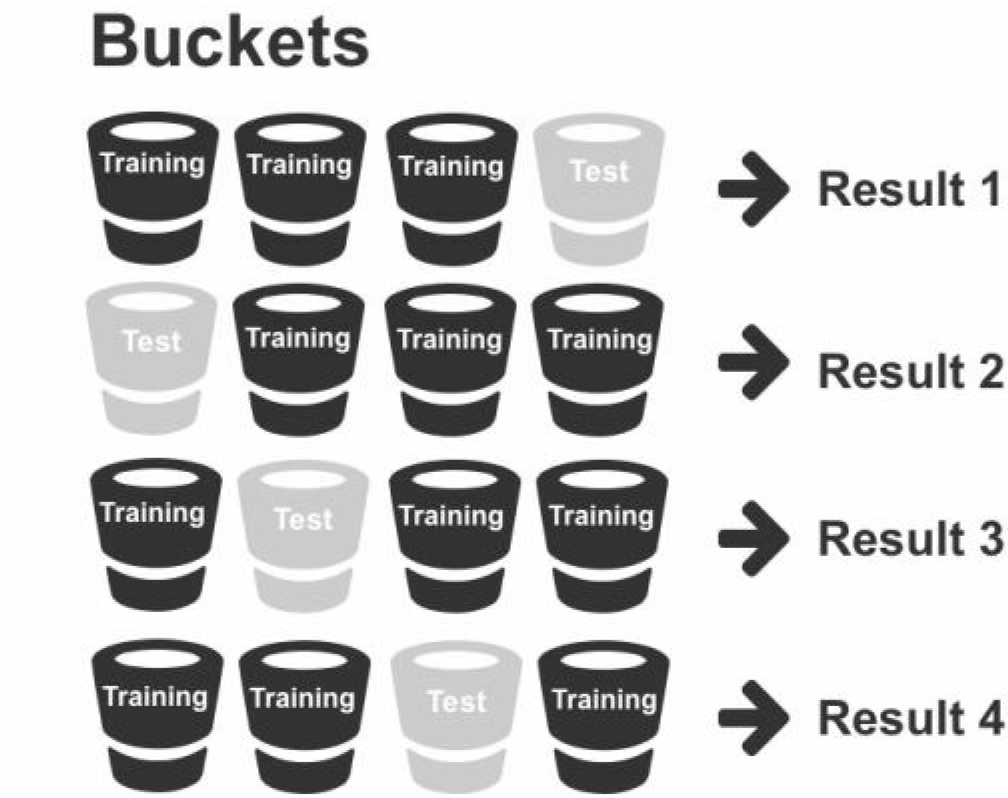


Figure 2: *k*-fold validation

The cross validation process is repeated *k* number of times ("folds"). At each fold, one bucket is reserved to test the training model generated by the other buckets. The process is repeated until all buckets have been utilized as both a

training and test bucket. The results are then aggregated and combined to formulate a single model.

By using all available data for both training and testing purposes, the *k*-fold validation technique dramatically minimizes potential error (such as overfitting) found by relying on a fixed split of training and test data.

How Much Data Do I Need?

A common question for students starting out in machine learning *is how much data do I need to train my dataset?* In general, machine learning works best when your training dataset includes a full range of feature combinations.

What does a full range of feature combinations look like? Imagine you have a dataset about data scientists categorized by the following features:

- University degree (X)
- 5+ years professional experience (X)
- Children (X)
- Salary (y)

To assess the relationship that the first three features (X) have to a data scientist's salary (y), we need a dataset that includes the y value for each combination of features. For instance, we need to know the salary for data scientists with a university degree, 5+ years professional experience and that don't have children, as well as data scientists with a university degree, 5+ years professional experience and that do have children.

The more available combinations, the more effective the model will be at capturing how each attribute affects y (the data scientist's salary). This will ensure that when it comes to putting the model into practice on the test data or real-life data, it won't immediately unravel at the sight of unseen combinations.

At a minimum, a machine learning model should typically have ten times as many data points as the total number of features. So for a small dataset with three features, the training data should ideally have at least thirty rows.

The other point to remember is that more relevant data is usually better than less. Having more relevant data allows you to cover more combinations and generally helps to ensure more accurate predictions. In some cases, it might not be possible or cost-effective to source data for every possible combination. In these cases, you will need to make do with the data that you have at your disposal.

The following chapters will examine specific algorithms commonly used in machine learning. Please note that I include some equations out of necessity, and I have tried to keep them as simple as possible. Many of the machine learning techniques that we discuss in this book already have working implementations in your programming language of choice—no equation writing necessary.

REGRESSION ANALYSIS

As the "Hello World" of machine learning algorithms, regression analysis is a simple supervised learning technique used to find the best trendline to describe a dataset.

The first regression analysis technique that we will examine is linear regression, which uses a straight line to describe a dataset. To unpack this simple technique, let's return to the earlier dataset charting Bitcoin values to the US Dollar.

Date	Bitcoin Price	No. of Days Transpired
19-05-2015	234.31	1
14-01-2016	431.76	240
09-07-2016	652.14	417
15-01-2017	817.26	607
24-05-2017	2358.96	736

Imagine you're back in high school and it's the year 2015 (which is probably much more recent than your actual year of graduation!). During your senior year, a news headline piques your interest in Bitcoin. With your natural tendency to chase the next shiny object, you tell your family about your cryptocurrency aspirations. But before you have a chance to bid for your first Bitcoin on Coinbase, your father intervenes and insists that you try paper trading before you go risking your life savings. "Paper trading" is using simulated means to buy and sell an investment without involving actual money.

So over the next twenty-four months, you track the value of Bitcoin and write down its value at regular intervals. You also keep a tally of how many days have passed since you first started paper trading. You never anticipated to still be paper trading after two years, but unfortunately, you never got a

chance to enter the cryptocurrency market. As suggested by your father, you waited for the value of Bitcoin to drop to a level you could afford. But instead, the value of Bitcoin exploded in the opposite direction.

Nonetheless, you haven't lost hope of one day owning Bitcoin. To assist your decision on whether you continue to wait for the value to drop or to find an alternative investment class, you turn your attention to statistical analysis. You first reach into your toolbox for a scatterplot. With the blank scatterplot in your hands, you proceed to plug in your x and y coordinates from your dataset and plot Bitcoin values from 2015 to 2017. However, rather than use all three columns from the table, you select the second (Bitcoin price) and third (No. of Days Transpired) columns to build your model and populate the scatterplot (shown in Figure 1). As we know, numerical values (found in the second and third columns) are easy to plug into a scatterplot and require no special conversion or one-hot encoding. What's more, the first and third columns contain the same variable of "time" and the third column alone is sufficient.

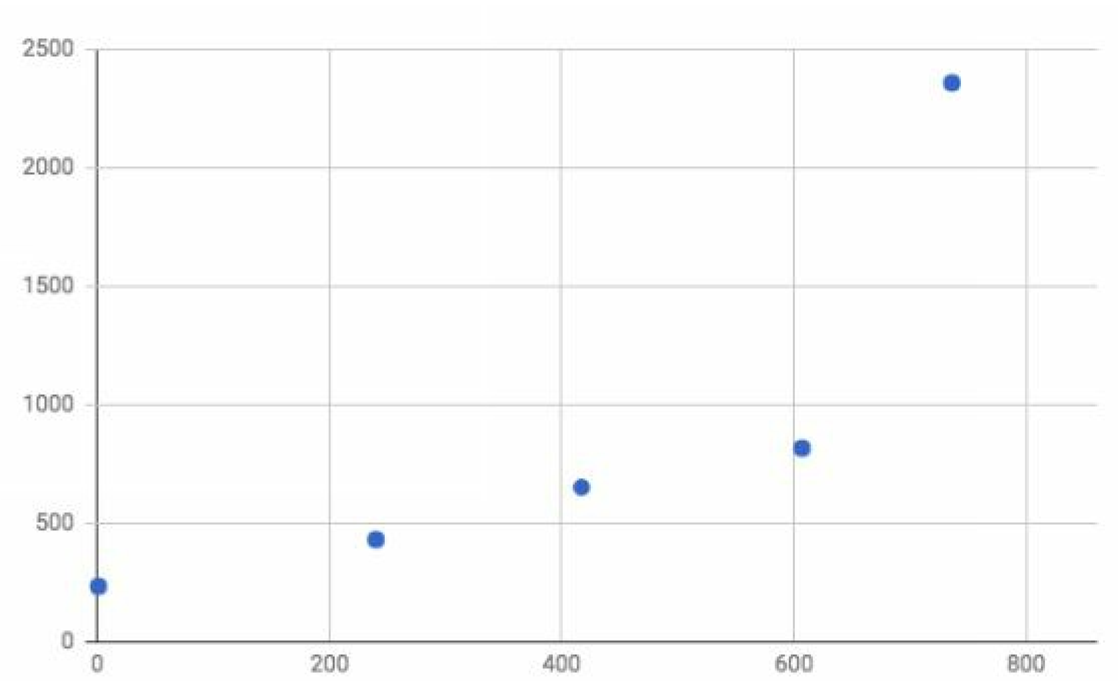


Figure 1: Bitcoin values from 2015-2017 plotted on a scatterplot

As your goal is to estimate what Bitcoin will be valued at in the future, the y-axis plots the dependent variable, which is "Bitcoin Price." The independent variable (X), in this case, is time. The "No. of Days Transpired" is thereby plotted on the x-axis.

After plotting the x and y values on the scatterplot, you can immediately see a trend in the form of a curve ascending from left to right with a steep increase between day 607 and day 736. Based on the upward trajectory of the curve, it might be time to quit hoping for a drop in value.

However, an idea suddenly pops up into your head. What if instead of waiting for the value of Bitcoin to fall to a level that you can afford, you instead borrow from a friend and purchase Bitcoin now at day 736? Then, when the value of Bitcoin rises further, you can pay back your friend and continue to earn asset appreciation on the Bitcoin you fully own.

In order to assess whether it's worth borrowing from your friend, you will need to first estimate how much you can earn in potential profit. Then you need to figure out whether the return on investment will be adequate to pay back your friend in the short-term.

It's now time to reach into the third compartment of the toolbox for an algorithm. One of the simplest algorithms in machine learning is regression analysis, which is used to determine the strength of a relationship between variables. Regression analysis comes in many forms, including linear, non-linear, logistic, and multilinear, but let's take a look first at linear regression.

Linear regression comprises a straight line that splits your data points on a scatterplot. The goal of linear regression is to split your data in a way that minimizes the distance between the regression line and all data points on the scatterplot. This means that if you were to draw a vertical line from the regression line to each data point on the graph, the aggregate distance of each point would equate to the smallest possible distance to the regression line.

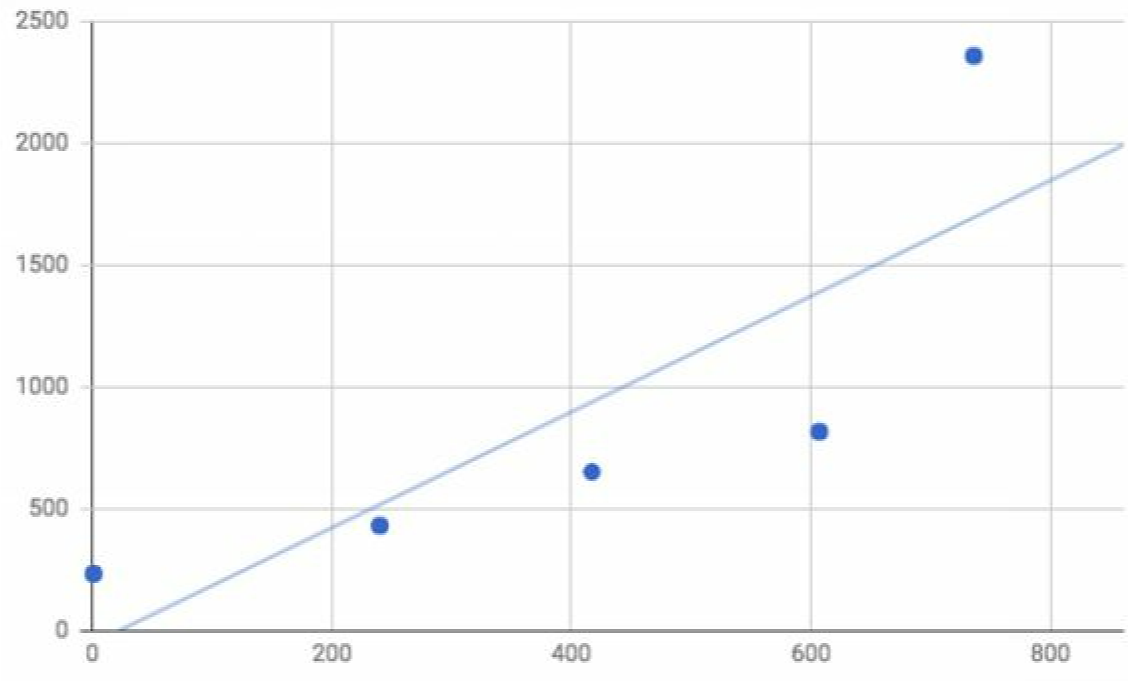


Figure 2: Linear regression line

The regression line is plotted on the scatterplot in Figure 2. The technical term for the regression line is the *hyperplane*, and you will see this term used throughout your study of machine learning. A hyperplane is practically a trendline—and this is precisely how Google Sheets titles linear regression in its scatterplot customization menu.

Another important feature of regression is *slope*, which can be conveniently calculated by referencing the hyperplane. As one variable increases, the other variable will increase at the average value denoted by the hyperplane. The slope is therefore very useful in formulating predictions. For example, if you wish to estimate the value of Bitcoin at 800 days, you can enter 800 as your x coordinate and reference the slope by finding the corresponding y value represented on the hyperplane. In this case, the y value is USD $1,850.

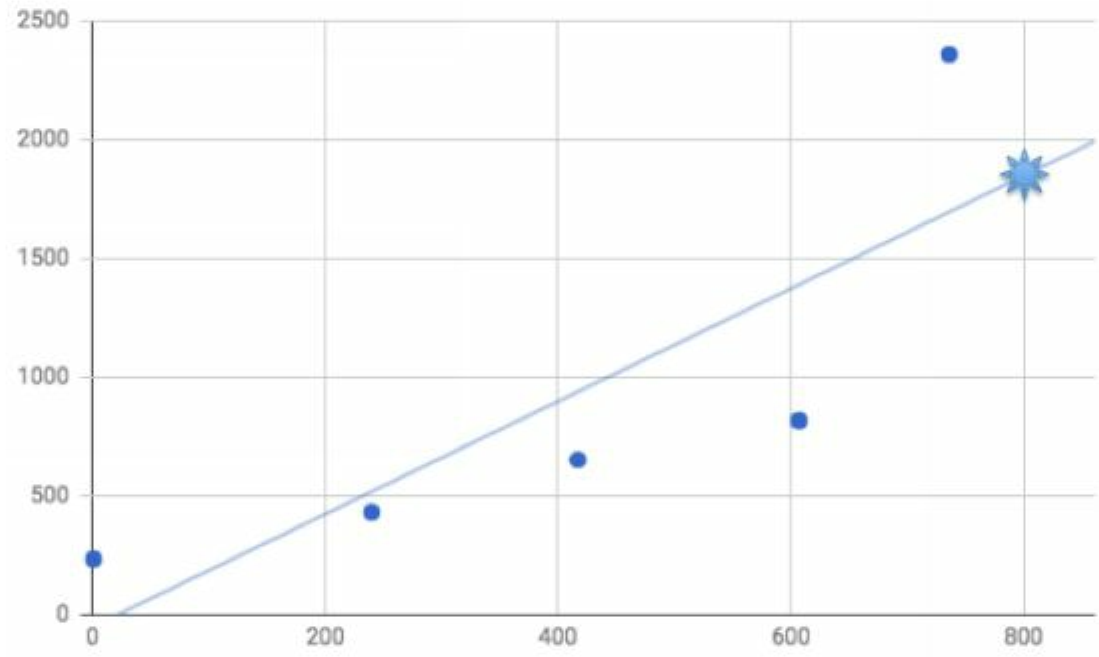


Figure 3: The value of Bitcoin at day 800

As shown in Figure 3, the hyperplane reveals that you actually stand to lose money on your investment at day 800 (after buying on day 736)! Based on the slope of the hyperplane, Bitcoin is expected to depreciate in value between day 736 and day 800—despite no precedent in your dataset for Bitcoin ever dropping in value.

While it’s needless to say that linear regression isn’t a fail-proof method to picking investment trends, the trendline does offer a basic reference point to predict the future. If we were to use the trendline as a reference point earlier in time, say at day 240, then the prediction posted would have been more accurate. At day 240 there is a low degree of deviation from the hyperplane, while at day 736 there is a high degree of deviation. Deviation refers to the distance between the hyperplane and the data point.

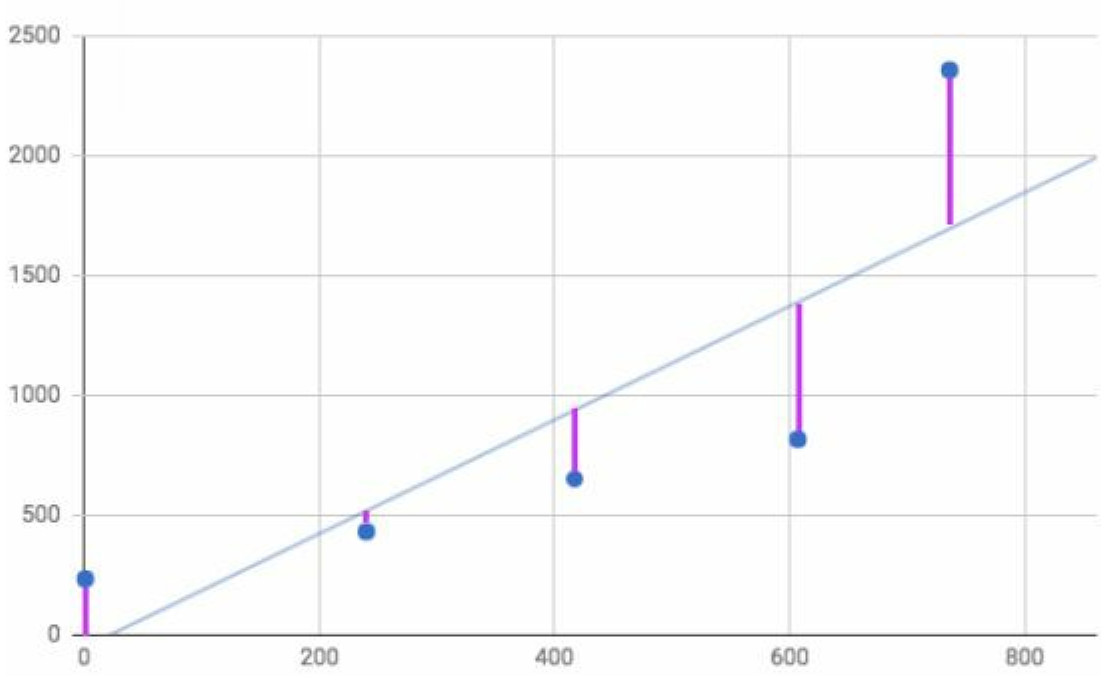


Figure 4: The distance of the data points to the hyperplane

In general, the closer the data points are to the regression line, the more accurate the final prediction. If there is a high degree of deviation between the data points and the regression line, the slope will provide less accurate predictions. Basing your predictions on the data point at day 736, where there is high deviation, results in poor accuracy. In fact, the data point at day 736 constitutes an outlier because it does not follow the same general trend as the previous four data points. What's more, as an outlier it exaggerates the trajectory of the hyperplane based on its high y-axis value. Unless future data points scale in proportion to the y-axis values of the outlier data point, the model's predictive accuracy will suffer.

Calculation Example

Although your programming language will take care of this automatically, it's useful to understand how linear regression is actually calculated. We will use the following dataset and formula to perform linear regression.

	(X)	(Y)	XY	X^2
1	1	3	3	1
2	2	4	8	4
3	1	2	2	1
4	4	7	28	16
5	3	5	15	9
Σ (Total)	**11**	**21**	**56**	**31**

The final two columns of the table are not part of the original dataset and have been added for convenience to complete the following equation.

$$a = \frac{(\Sigma y)(\Sigma x^2) - (\Sigma x)(\Sigma xy)}{n(\Sigma x^2) - (\Sigma x)^2}$$

$$b = \frac{n(\Sigma xy) - (\Sigma x)(\Sigma y)}{n(\Sigma x^2) - (\Sigma x)^2}$$

Where:

Σ = Total sum

Σx = Total sum of all x values (1 + 2 + 1 + 4 + 3 = 11)

Σy = Total sum of all y values (3 + 4 + 2 + 7 + 5 = 21)

Σxy = Total sum of x*y for each row (3 + 8 + 2 + 28 + 15 = 56)

Σx^2 = Total sum of x*x for each row (1 + 4 + 1 + 16 + 9 = 31)

n = Total number of rows. In the case of this example, n = 5.

$$a = \frac{(\Sigma y)(\Sigma x^2) - (\Sigma x)(\Sigma xy)}{n(\Sigma x^2) - (\Sigma x)^2} \qquad a = \frac{(21)(31) - (11)(56)}{5(31) - (11)^2}$$

$$b = \frac{n(\Sigma xy) - (\Sigma x)(\Sigma y)}{n(\Sigma x^2) - (\Sigma x)^2} \qquad b = \frac{5(56) - (11)(21)}{5(31) - (11)^2}$$

A =
((21 x 31) – (11 x 56)) / (5(31) – 11^2)
(651 – 616) / (155 – 121)
35 / 34
1.029

B =
(5(56) – (11 x 21)) / (5(31) – 11^2)
(280 – 231) / (155 – 121)
49 / 34
1.44
Insert the “a” and “b” values into a linear equation.
y = a + bx
y = 1.029 + 1.441x
The linear equation y = 1.029 + 1.441x dictates how to draw the hyperplane.

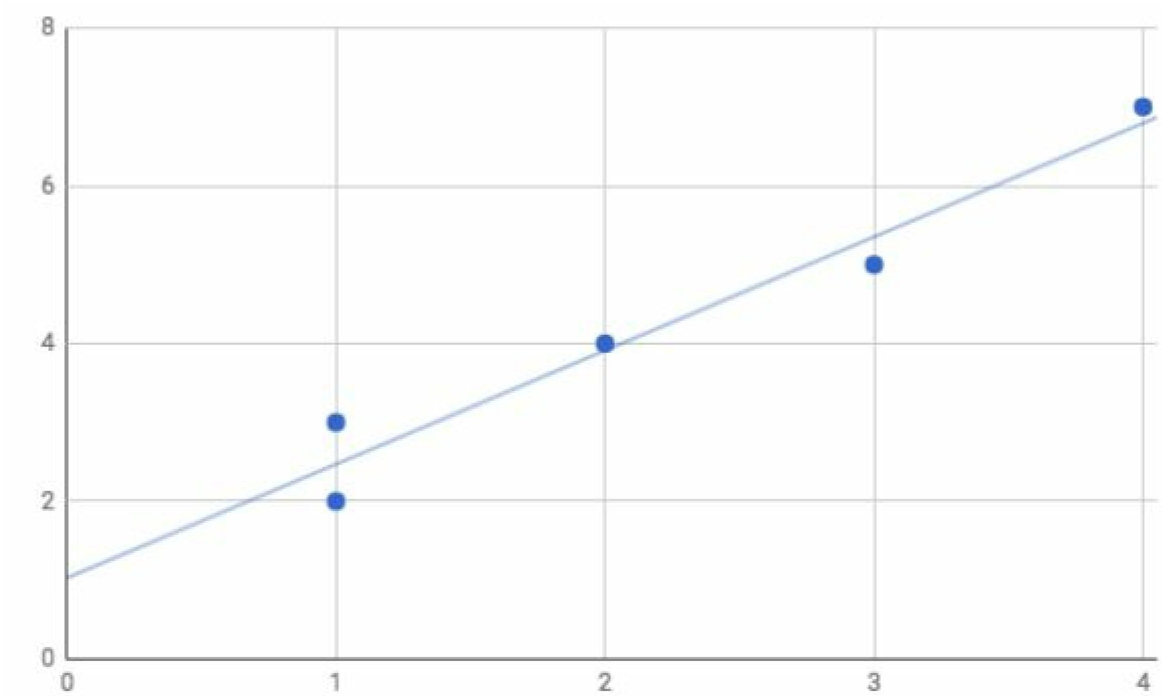


Figure 5: The linear regression hyperplane plotted on the scatterplot

Let's now test the regression line by looking up the coordinates for x = 2.
y = 1.029 + 1.441(x)
y = 1.029 + 1.441(2)
y = 3.911
In this case, the prediction is very close to the actual result of 4.0.

Logistic Regression

A large part of data analysis boils down to a simple question: is something "A" or "B?" Is it "positive" or "negative?" Is this person a "potential customer" or "not a potential customer?" Machine learning accommodates such questions through logistic equations, and specifically through what is known as the *sigmoid function*. The sigmoid function produces an S-shaped curve that can convert any number and map it into a numerical value between 0 and 1, but it does so without ever reaching those exact limits.

A common application of the sigmoid function is found in logistic regression. Logistic regression adopts the sigmoid function to analyze data and predict discrete classes that exist in a dataset. Although logistic regression shares a visual resemblance to linear regression, it is technically a classification technique. Whereas linear regression addresses numerical equations and forms numerical predictions to discern relationships between variables,

logistic regression predicts discrete classes.

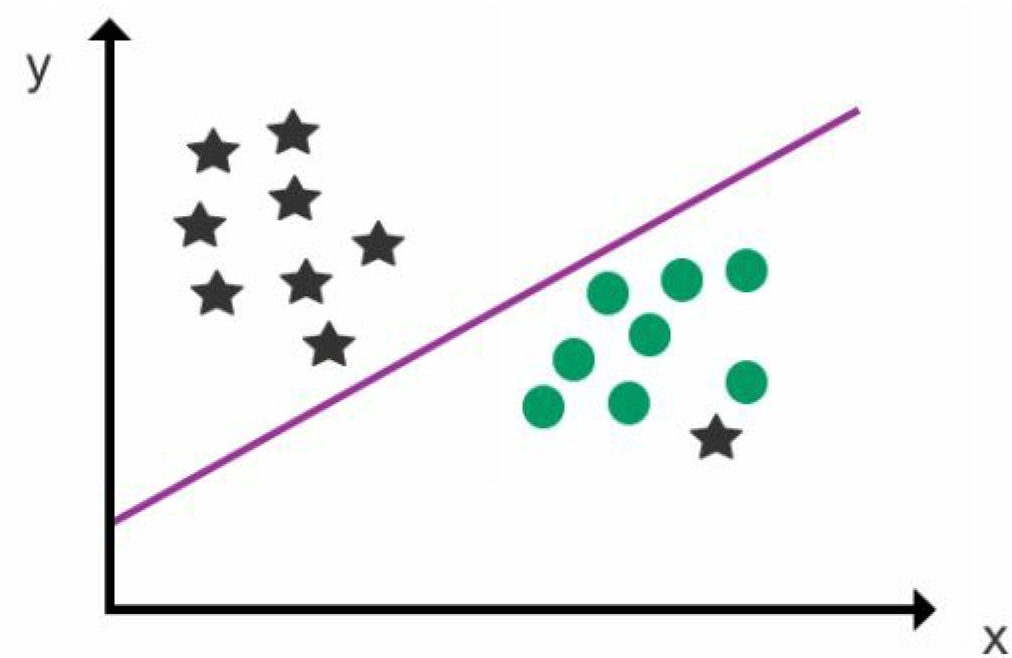


Figure 6: An example of logistic regression

Logistic regression is typically used for binary classification to predict two discrete classes, e.g. *pregnant* or *not pregnant*. To do this, the sigmoid function (shown as follows) is added to compute the result and convert numerical results into an expression of probability between 0 and 1.

$$y = \frac{1}{1+e^{-x}}$$

The logistic sigmoid function above is calculated as “1” divided by “1” plus “e” raised to the power of negative “x,” where:

x = the numerical value you wish to transform

e = Euler's constant, 2.718

In a binary case, a value of 0 represents no chance of occurring, and 1 represents a certain chance of occurring. The degree of probability for values located between 0 and 1 can be calculated according to how close they rest to 0 (impossible) or 1 (certain possibility) on the scatterplot.

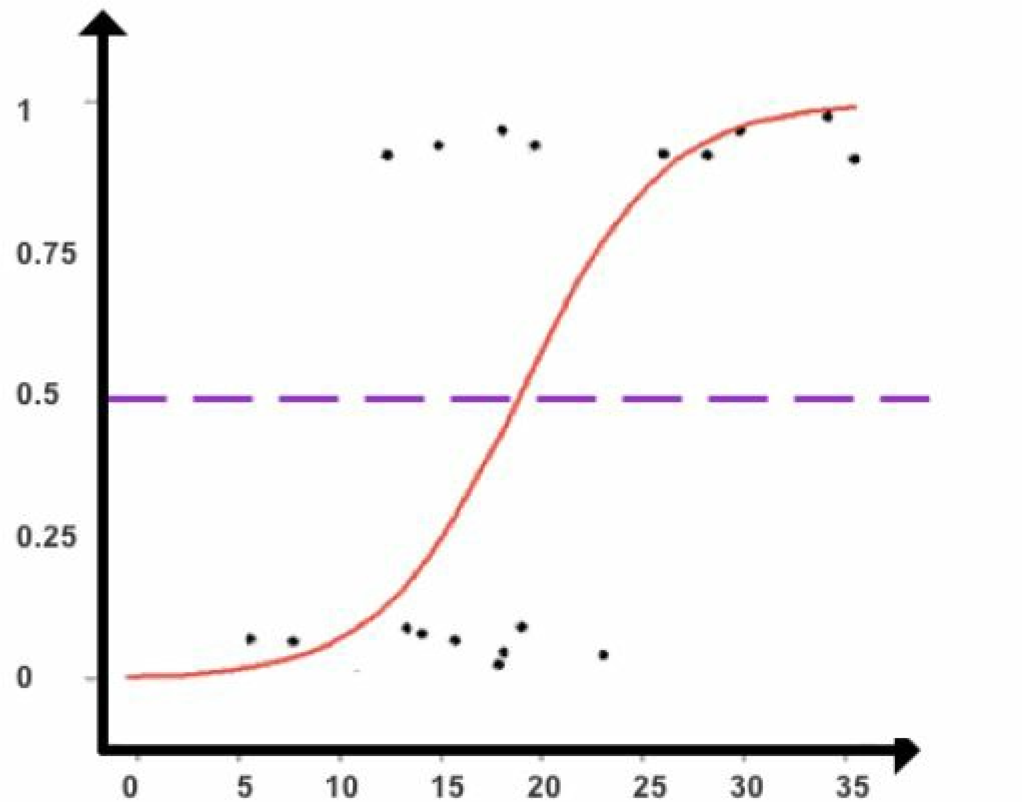


Figure 7: A sigmoid function used to classify data points

Based on the found probabilities we can assign each data point to one of two discrete classes. As seen in Figure 7, we can create a cut-off point at 0.5 to classify the data points into classes. Data points that record a value above 0.5 are classified as Class A, and any data points below 0.5 are classified as Class B. Data points that record a result of exactly 0.5 are unclassifiable, but such instances are rare due to the mathematical component of the sigmoid function.

Please also note that this formula alone does not produce the hyperplane dividing discrete categories as seen earlier in Figure 6. The statistical formula for plotting the logistic hyperplane is somewhat more complicated and can be conveniently plotted using your programming language.

Given its strength in binary classification, logistic regression is used in many fields including fraud detection, disease diagnosis, emergency detection, loan default detection, or to identify spam email through the process of identifying specific classes, e.g. non-spam and spam. However, logistic regression can also be applied to ordinal cases where there are a set number of discrete values, e.g. single, married, and divorced.

Logistic regression with more than two outcome values is known as

multinomial logistic regression, which can be seen in Figure 8.

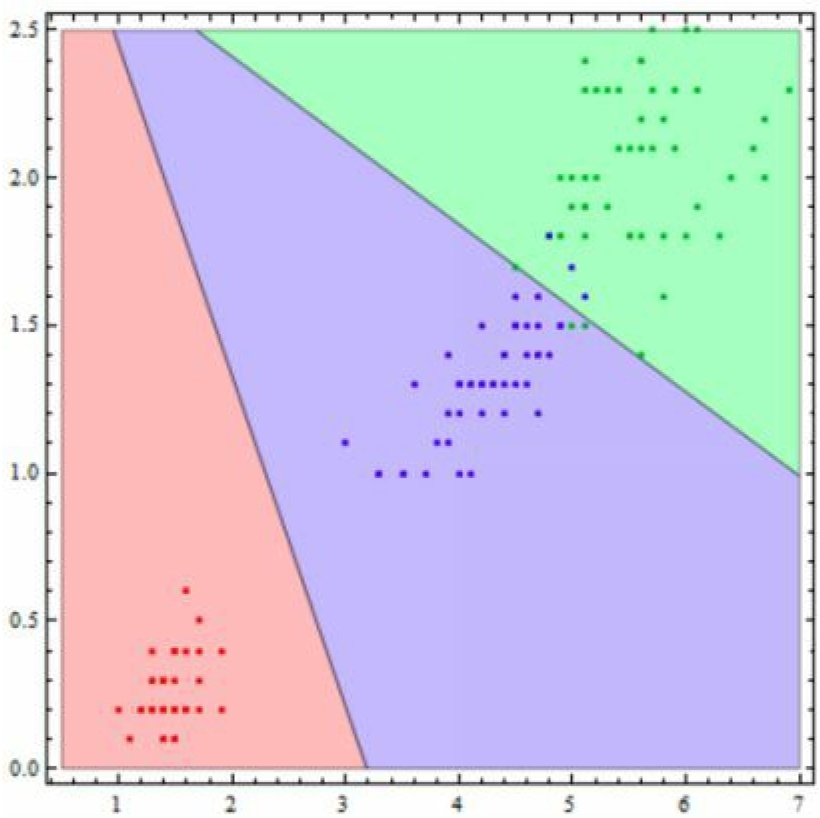


Figure 8: An example of multinomial logistic regression

Two tips to remember when performing logistic regression are that the data should be free of missing values and that all variables are independent of each other. There should also be sufficient data for each outcome value to ensure high accuracy. A good starting point would be approximately 30-50 data points for each outcome, i.e. 60-100 total data points for binary logistic regression.

Support Vector Machine

As an advanced category of regression, support vector machine (SVM) resembles logistic regression but with stricter conditions. To that end, SVM is superior at drawing classification boundary lines. Let’s examine what this looks like in action.

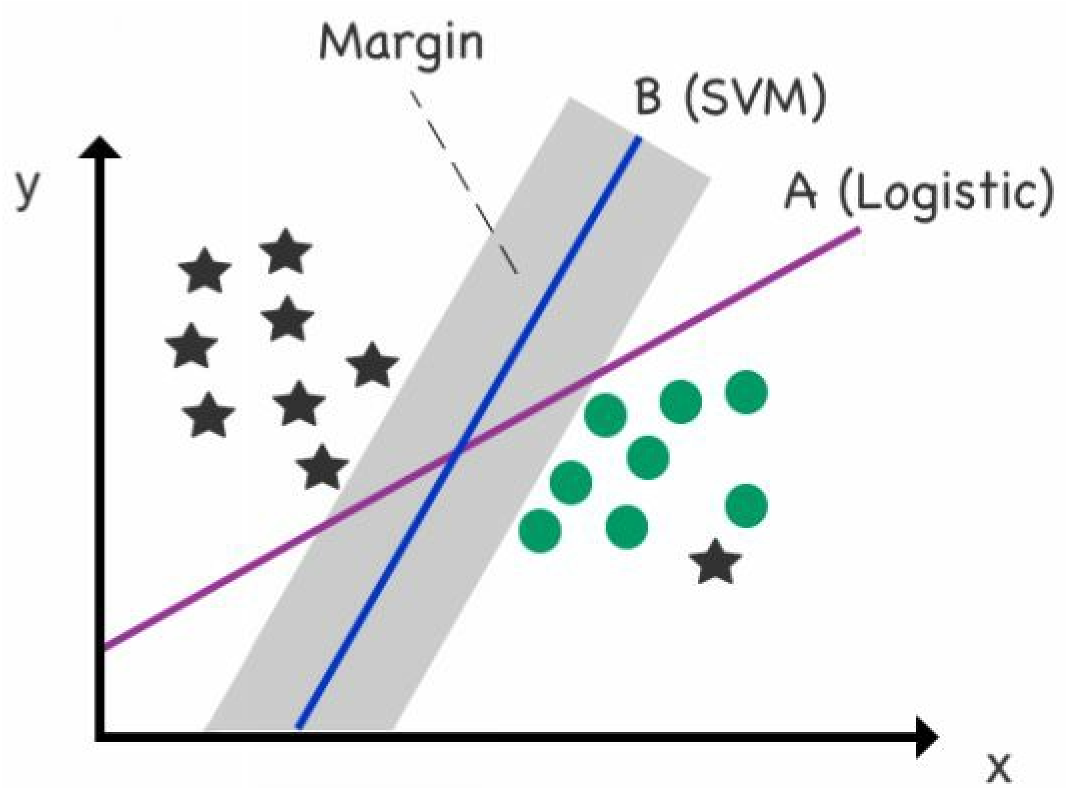


Figure 9: Logistic regression versus SVM

The scatterplot in Figure 9 consists of data points that are linearly separable and the logistic hyperplane (A) splits the data points into two classes in a way that minimizes the distance between all data points and the hyperplane. The second line, the SVM hyperplane (B), likewise separates the two clusters, but from a position of maximum distance between itself and the two clusters.

You will also notice a gray area that denotes *margin*, which is the distance between the hyperplane and the nearest data point, multiplied by two. The margin is a key feature of SVM and is important because it offers additional support to cope with new data points that may infringe on a logistic regression hyperplane. To illustrate this scenario, let’s consider the same scatterplot with the inclusion of a new data point.

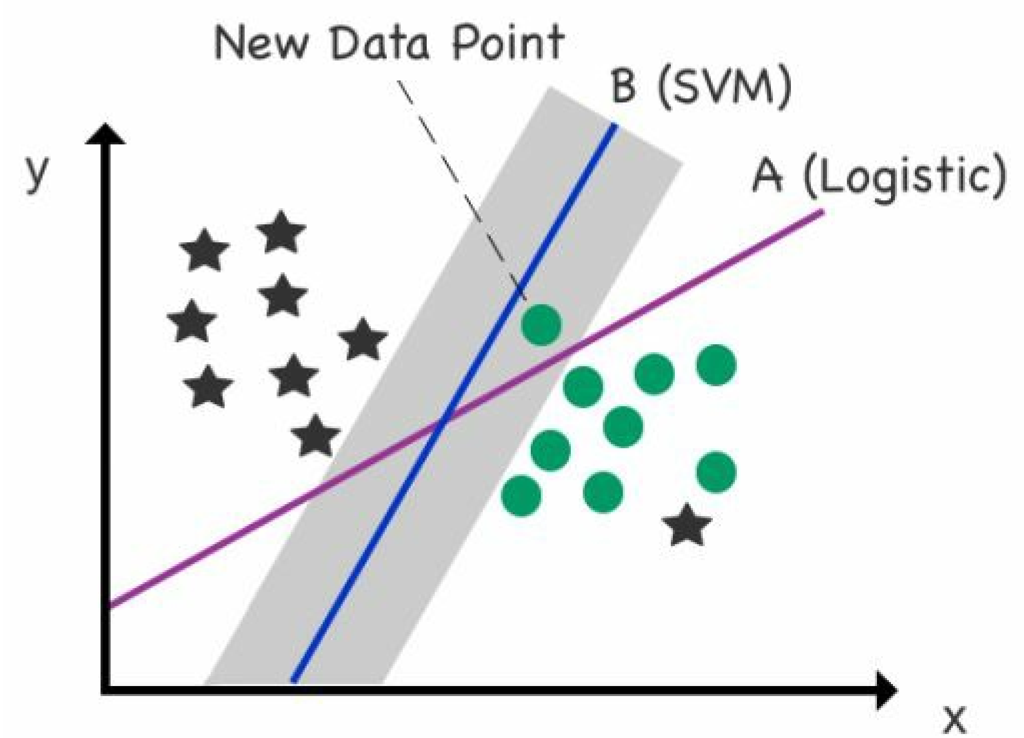


Figure 10: A new data point is added to the scatterplot

The new data point is a circle, but it is located incorrectly on the left side of the logistic regression hyperplane (designated for stars). The new data point, though, remains correctly located on the right side of the SVM hyperplane (designated for circles) courtesy of ample “support” supplied by the margin.

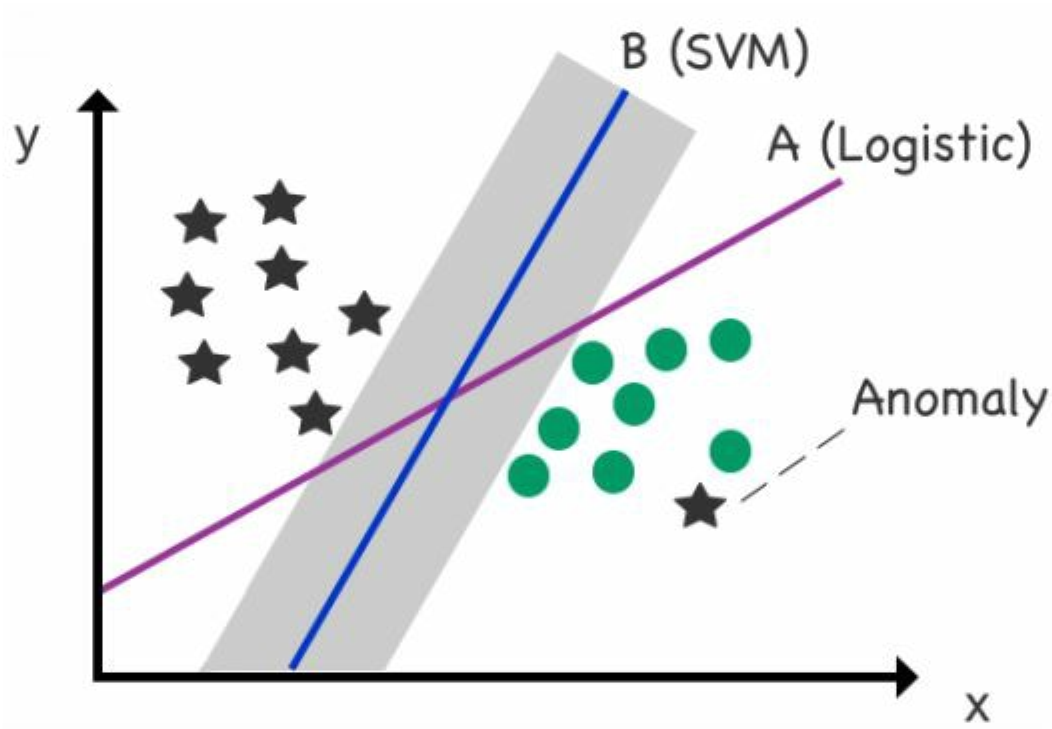


Figure 11: Mitigating anomalies

Another useful application case of SVM is for mitigating anomalies. A limitation of standard logistic regression is that it goes out of its way to fit anomalies (as seen in the scatterplot with the star in the bottom right corner in Figure 11). SVM, however, is less sensitive to such data points and actually minimizes their impact on the final location of the boundary line. In Figure 11, we can see that Line B (SVM hyperplane) is less sensitive to the anomalous star on the right-hand side. SVM can thus be used as one method to fight anomalies.

The examples seen so far have comprised two features plotted on a two-dimensional scatterplot. However, SVM's real strength is found in high-dimensional data and handling multiple features. SVM has numerous variations available to classify high-dimensional data, known as "kernels," including linear SVC (seen in Figure 12), polynomial SVC, and the Kernel Trick. The Kernel Trick is an advanced solution to map data from a low-dimensional to a high-dimensional space. Transitioning from a two-dimensional to a three-dimensional space allows you to use a linear plane to split the data within a 3-D space, as seen in Figure 12.

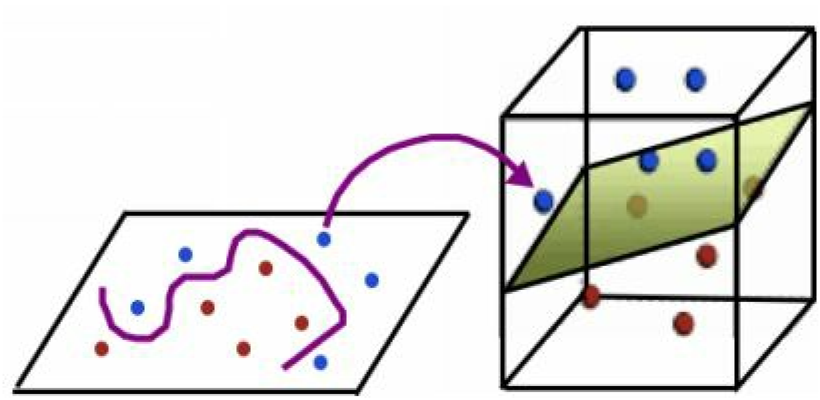


Figure 12: Example of linear SVC

CLUSTERING

One helpful approach to analyze information is to identify clusters of data that share similar attributes. For example, your company may wish to examine a segment of customers that purchase at the same time of the year and discern what factors influence their purchasing behavior.

By understanding a particular cluster of customers, you can form decisions about which products to recommend to customer groups through promotions and personalized offers. Outside of market research, clustering can be applied to various other scenarios, including pattern recognition, fraud detection, and image processing.

Clustering analysis falls under the banner of both supervised learning and unsupervised learning. As a supervised learning technique, clustering is used to classify new data points into existing clusters through *k*-nearest neighbors (*k*-NN) and as an unsupervised learning technique, clustering is applied to identify discrete groups of data points through *k*-means clustering. Although there are other forms of clustering techniques, these two algorithms are generally the most popular in both machine learning and data mining.

k-Nearest Neighbors

The simplest clustering algorithm is *k*-nearest neighbors (*k*-NN); a supervised learning technique used to classify new data points based on the relationship to nearby data points.

k-NN is similar to a voting system or a popularity contest. Think of it as being the new kid in school and choosing a group of classmates to socialize with based on the five classmates who sit nearest to you. Among the five classmates, three are **geeks**, one is a **skater,** and one is a **jock**. According to *k*-NN, you would choose to hang out with the **geeks** based on their numerical advantage. Let's look at another example.

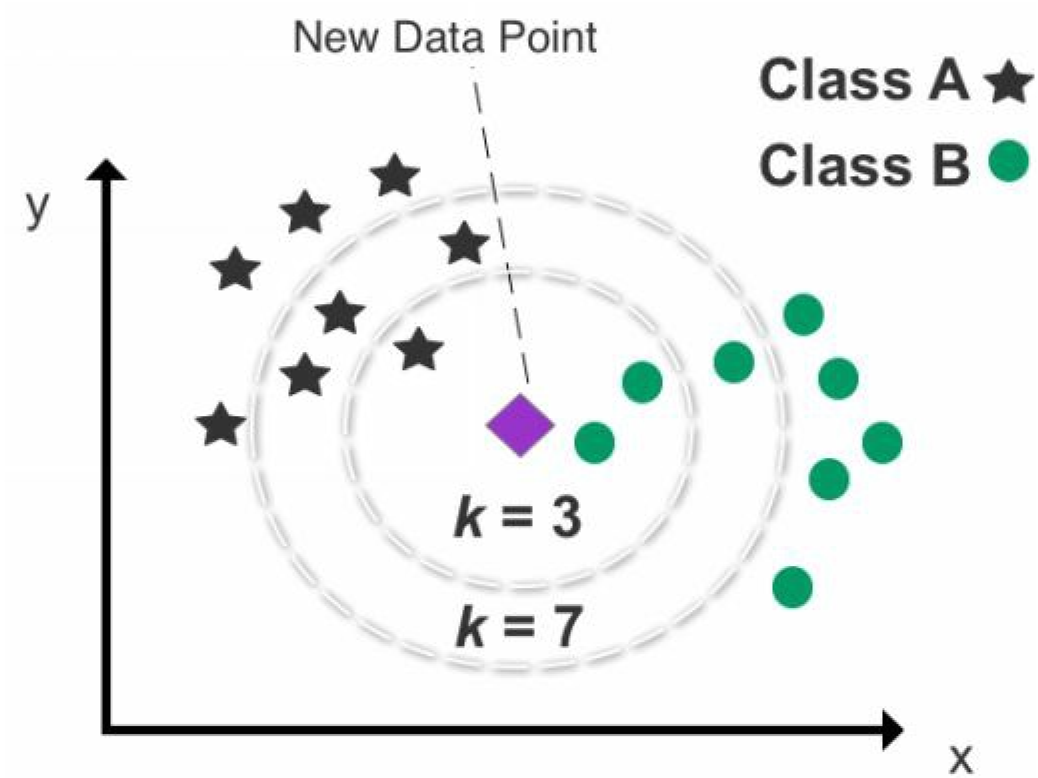


Figure 1: An example of *k*-NN clustering used to predict the class of a new data point

As seen in Figure 1, the scatterplot enables us to compute the distance between any two data points. The data points on the scatterplot have already been categorized into two clusters. Next, a new data point whose class is unknown is added to the plot. We can predict the category of the new data point based on its relationship to existing data points.

First though, we must set "*k*" to determine how many data points we wish to nominate to classify the new data point. If we set *k* to 3, *k*-NN will only analyze the new data point's relationship to the three closest data points (neighbors). The outcome of selecting the three closest neighbors returns two Class B data points and one Class A data point. Defined by *k* (3), the model's prediction for determining the category of the new data point is Class B as it returns two out of the three nearest neighbors.

The chosen number of neighbors identified, defined by *k*, is crucial in determining the results. In Figure 1, you can see that classification will change depending on whether *k* is set to "3" or "7." It is therefore recommended that you test numerous *k* combinations to find the best fit and avoid setting *k* too low or too high. Setting *k* to an uneven number will also help to eliminate the possibility of a statistical stalemate and invalid result. The default number of neighbors is five when using Scikit-learn.

Although generally a highly accurate and simple technique to learn, storing an entire dataset and calculating the distance between each new data point and all existing data points does place a heavy burden on computing resources. Thus, *k*-NN is generally not recommended for use with large datasets.

Another potential downside is that it can be challenging to apply *k*-NN to high-dimensional data (3-D and 4-D) with multiple features. Measuring multiple distances between data points in a three or four-dimensional space is taxing on computing resources and also complicated to perform accurate classification. Reducing the total number of dimensions, through a descending dimension algorithm such as Principle Component Analysis (PCA) or merging variables, is a common strategy to simplify and prepare a dataset for *k*-NN analysis.

k-Means Clustering

As a popular unsupervised learning algorithm, *k*-means clustering attempts to divide data into *k* discrete groups and is effective at uncovering basic data patterns. Examples of potential groupings include animal species, customers with similar features, and housing market segmentation. The *k*-means clustering algorithm works by first splitting data into *k* number of clusters with *k* representing the number of clusters you wish to create. If you choose to split your dataset into three clusters then *k*, for example, is set to 3.

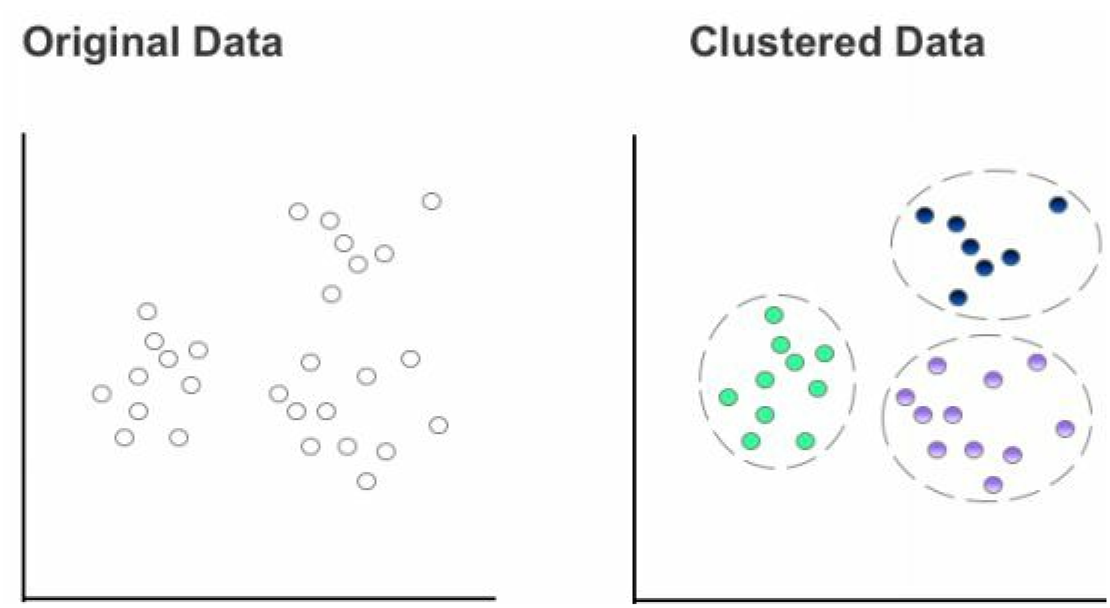


Figure 2: Comparison of original data and clustered data using *k*-means

In Figure 2, we can see that the original (unclustered) data has been transformed into three clusters (*k* is 3). If we were to set *k* to 4, an additional cluster would be derived from the dataset to produce four clusters.

How does *k*-means clustering separate the data points? The first step is to examine the unclustered data on the scatterplot and manually select a centroid for each *k* cluster. That centroid then forms the epicenter of an individual cluster. Centroids can be chosen at random, which means you can nominate any data point on the scatterplot to act as a centroid. However, you can save time by choosing centroids dispersed across the scatterplot and not directly adjacent to each other. In other words, start by guessing where you think the centroids for each cluster might be located. The remaining data points on the scatterplot are then assigned to the closest centroid by measuring the Euclidean distance.

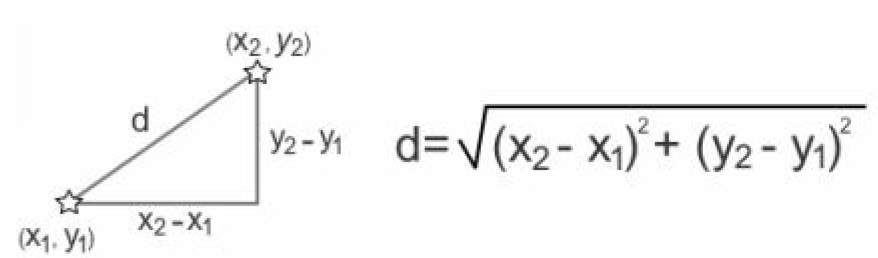


Figure 3: Calculating Euclidean distance

Each data point can be assigned to only one cluster and each cluster is discrete. This means that there is no overlap between clusters and no case of nesting a cluster inside another cluster. Also, all data points, including anomalies, are assigned to a centroid irrespective of how they impact the final shape of the cluster. However, due to the statistical force that pulls all nearby data points to a central point, your clusters will generally form an elliptical or spherical shape.

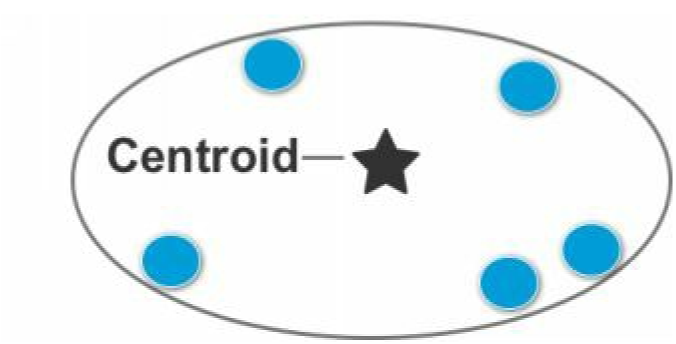


Figure 4: Example of an ellipse cluster

After all data points have been allocated to a centroid, the next step is to aggregate the mean value of all data points for each cluster, which can be found by calculating the average x and y values of all data points in that cluster.

Next, take the mean value of the data points in each cluster and plug in those x and y values to update your centroid coordinates. This will most likely result in a change to your centroids' location. Your total number of clusters, however, will remain the same. You are not creating new clusters, rather updating their position on the scatterplot. Like musical chairs, the remaining data points will then rush to the closest centroid to form *k* number of clusters. Should any data point on the scatterplot switch clusters with the changing of centroids, the previous step is repeated. This means, again, calculating the average mean value of the cluster and updating the x and y values of each centroid to reflect the average coordinates of the data points in that cluster.

Once you reach a stage where the data points no longer switch clusters after an update in centroid coordinates, the algorithm is complete, and you have your final set of clusters. The following diagrams break down the full algorithmic process.

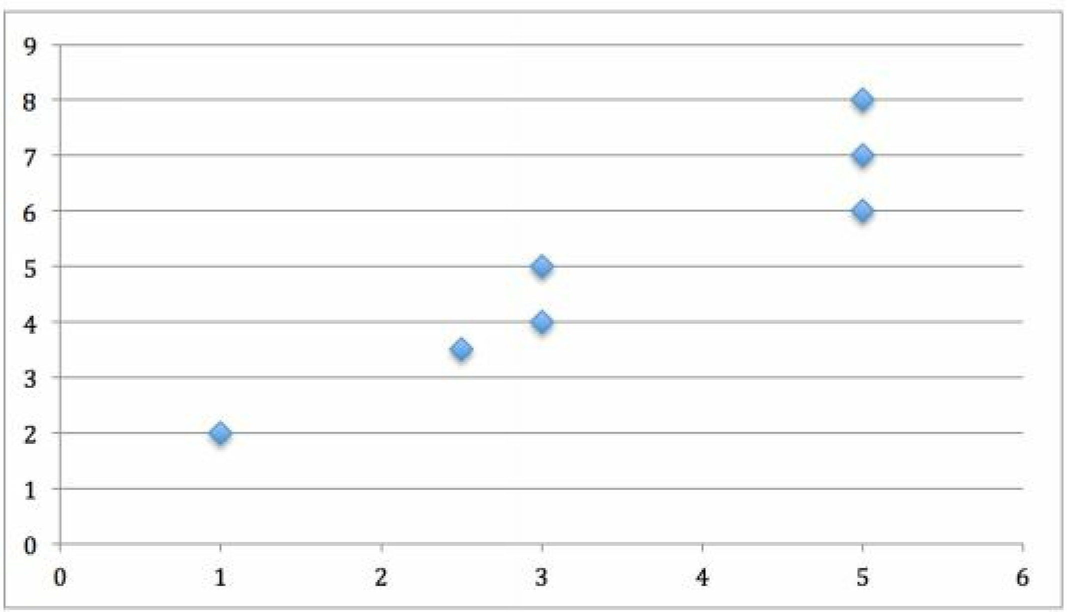


Figure 5: Sample data points are plotted on a scatterplot

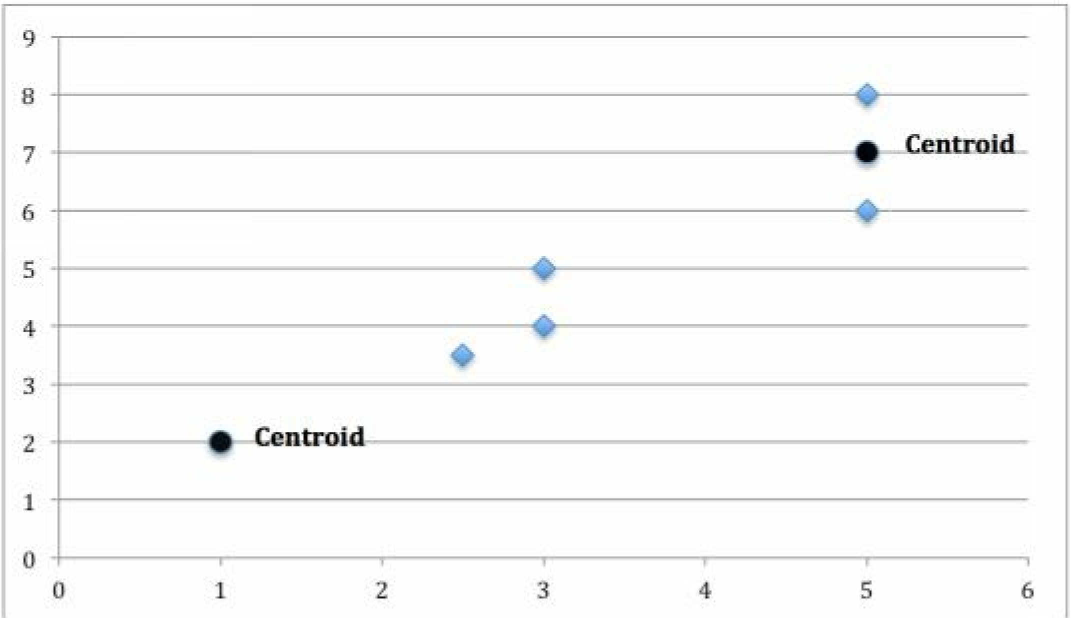


Figure 6: Two data points are nominated as centroids

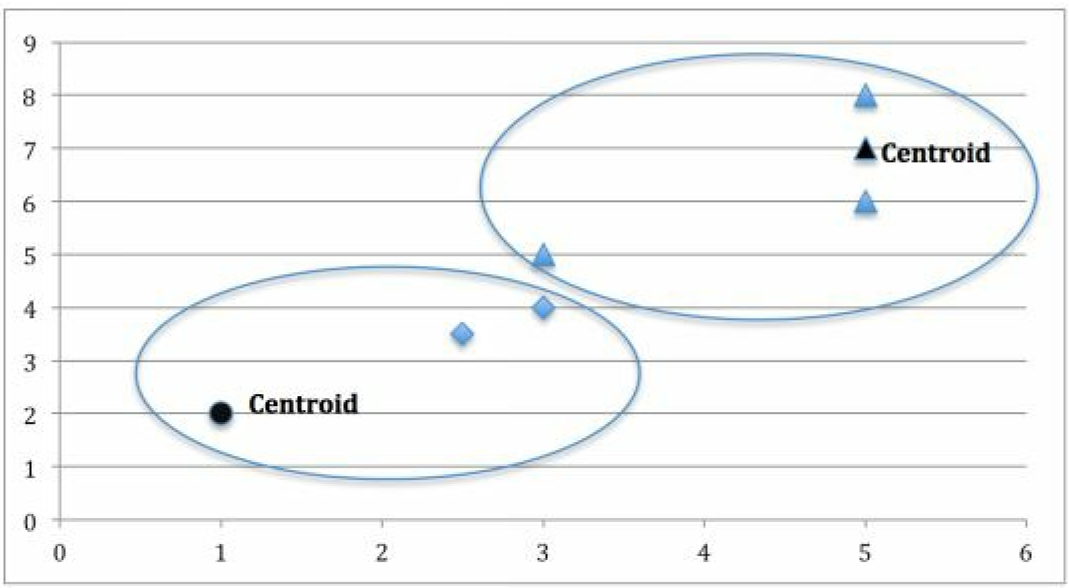


Figure 7: Two clusters are formed after calculating the Euclidean distance of the remaining data points to the centroids.

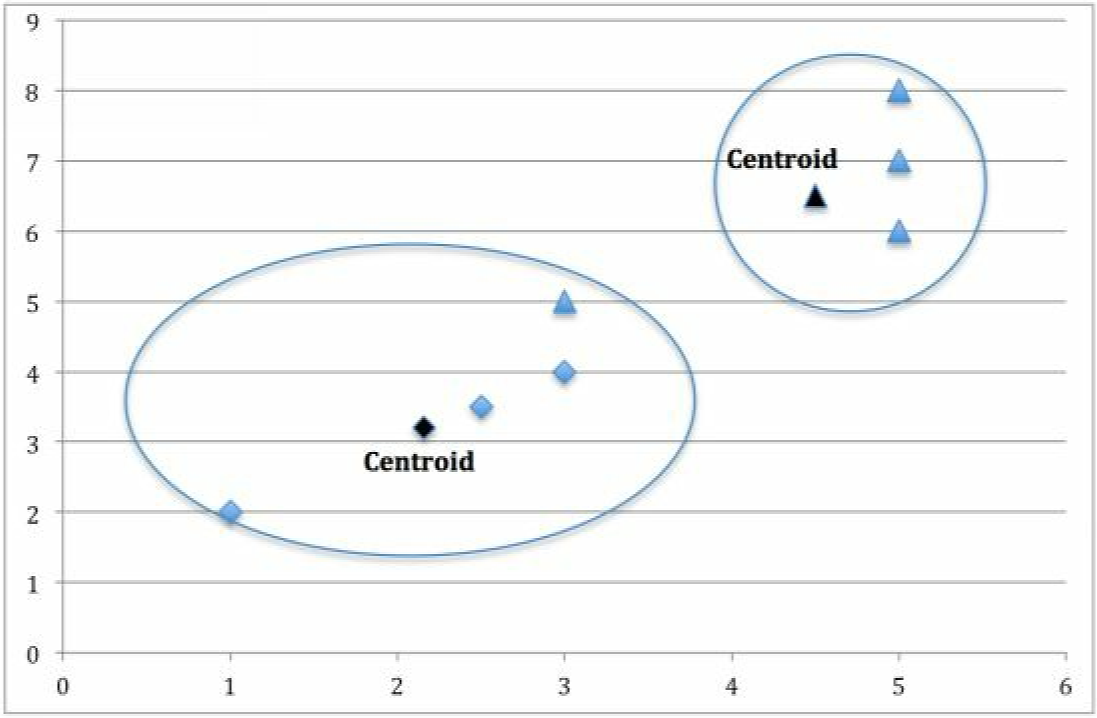


Figure 8: The centroid coordinates for each cluster are updated to reflect the cluster's mean value. As one data point has switched from the right cluster to the left cluster, the centroids of both clusters are recalculated.

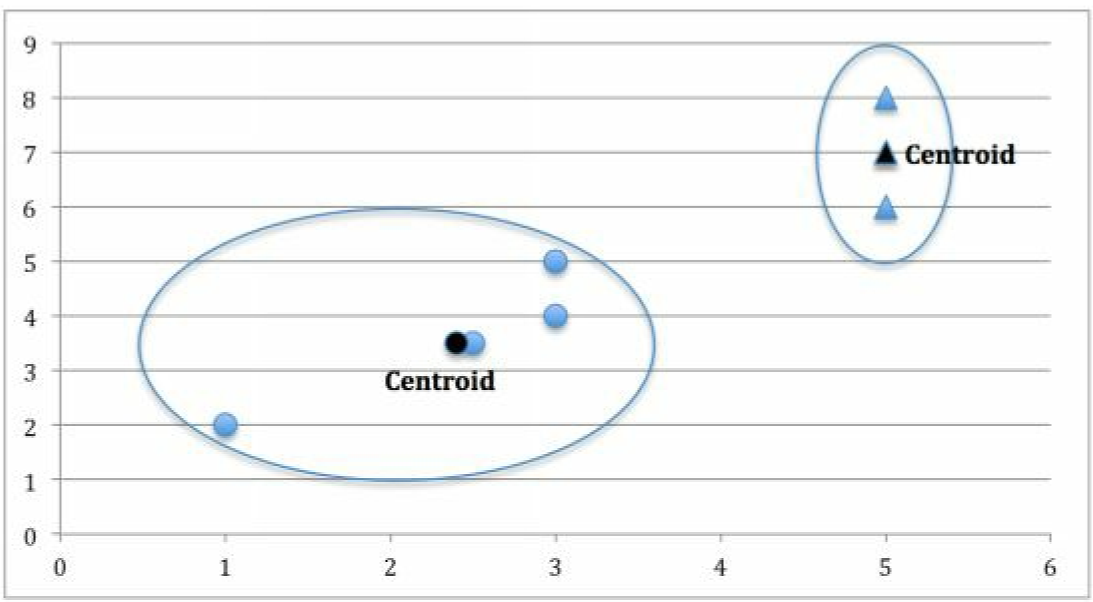


Figure 9: Two final clusters are produced based on the updated centroids for each cluster

Setting *k*

In setting *k*, it is important to strike the right number of clusters. In general, as *k* increases, clusters become smaller and variance falls. However, the downside is that neighboring clusters become less distinct from one another as *k* increases.

If you set *k* to the same number of data points in your dataset, each data point automatically converts into a standalone cluster. Conversely, if you set *k* to 1, then all data points will be deemed as homogenous and produce only one cluster. Needless to say, setting *k* to either extreme will not provide any worthy insight to analyze.

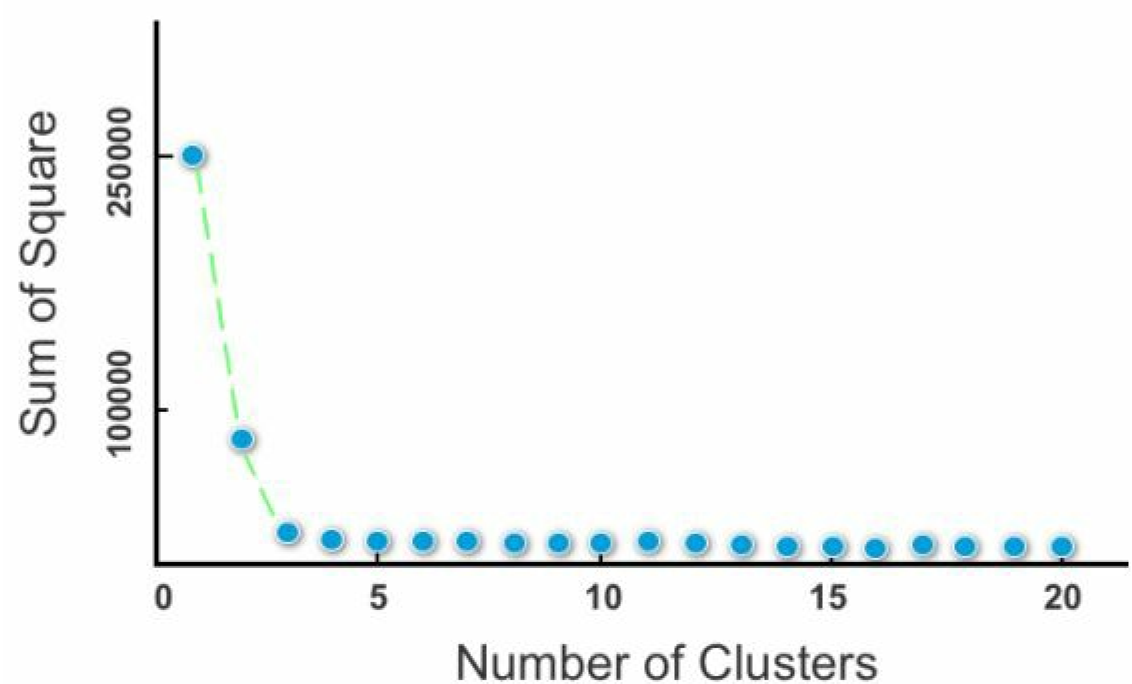


Figure 10: A scree plot

In order to optimize *k*, you may wish to turn to a scree plot for guidance. A scree plot charts the degree of scattering (variance) inside a cluster as the total number of clusters increase. Scree plots are famous for their iconic "elbow," which reflects several pronounced kinks in the plot's curve.

A scree plot compares the Sum of Squared Error (SSE) for each variation of total clusters. SSE is measured as the sum of the squared distance between the centroid and the other neighbors inside the cluster. In a nutshell, SSE drops as more clusters are formed.

This then raises the question of what the optimal number of clusters is. In general, you should opt for a cluster solution where SSE subsides dramatically to the left on the scree plot, but before it reaches a point of negligible change with cluster variations to its right. For instance, in Figure 10, there is little impact on SSE for six or more clusters. This would result in clusters that would be small and difficult to distinguish.

In this scree plot, two or three clusters appear to be an ideal solution. There

exists a significant kink to the left of these two cluster variations due to a pronounced drop-off in SSE. Meanwhile, there is still some change in SSE with the solution to their right. This will ensure that these two cluster solutions are distinct and have an impact on data classification.

A more simple and non-mathematical approach to setting *k* is applying domain knowledge. For example, if I am analyzing data concerning visitors to the website of a major IT provider, I might want to set *k* to 2. Why two clusters? Because I already know there is likely to be a major discrepancy in spending behavior between returning visitors and new visitors. First-time visitors rarely purchase enterprise-level IT products and services, as these customers will normally go through a lengthy research and vetting process before procurement can be approved.

Hence, I can use *k*-means clustering to create two clusters and test my hypothesis. After creating two clusters, I may then want to examine one of the two clusters further, either applying another technique or again using *k*-means clustering. For example, I might want to split returning users into two clusters (using *k*-means clustering) to test my hypothesis that mobile users and desktop users produce two disparate groups of data points. Again, by applying domain knowledge, I know it is uncommon for large enterprises to make big-ticket purchases on a mobile device. Still, I wish to create a machine learning model to test this assumption.

If, though, I am analyzing a product page for a low-cost item, such as a $4.99 domain name, new visitors and returning visitors are less likely to produce two clear clusters. As the product item is of low value, new users are less likely to deliberate before purchasing.

Instead, I might choose to set *k* to 3 based on my three primary lead generators: organic traffic, paid traffic, and email marketing. These three lead sources are likely to produce three discrete clusters based on the facts that:

a) **Organic traffic** generally consists of both new and returning customers with a strong intent of purchasing from my website (through pre-selection, e.g. word of mouth, previous customer experience).

b) **Paid traffic** targets new customers who typically arrive on the website with a lower level of trust than organic traffic, including potential customers who click on the paid advertisement by mistake.

c) **Email marketing** reaches existing customers who already have experience purchasing from the website and have established user accounts.

This is an example of domain knowledge based on my own occupation, but do understand that the effectiveness of "domain knowledge" diminishes dramatically past a low number of *k* clusters. In other words, domain knowledge might be sufficient for determining two to four clusters, but it will be less valuable in choosing between 20 or 21 clusters.

BIAS & VARIANCE

Algorithm selection is an important step in forming an accurate prediction model, but deploying an algorithm with a high rate of accuracy can be a difficult balancing act. The fact that each algorithm can produce vastly different models based on the hyperparameters provided can lead to dramatically different results. As mentioned earlier, hyperparameters are the algorithm's settings, similar to the controls on the dashboard of an airplane or the knobs used to tune radio frequency—except hyperparameters are lines of code!

```
model = ensemble.GradientBoostingRegressor(
    n_estimators=150,
    learning_rate=0.1,
    max_depth=4,
    min_samples_split=4,
    min_samples_leaf=4,
    max_features=0.5,
    loss='huber'
)
```

Figure 1: Example of hyperparameters in Python for the algorithm gradient boosting

A constant challenge in machine learning is navigating *underfitting* and *overfitting*, which describe how closely your model follows the actual patterns of the dataset. To understand underfitting and overfitting, you must first understand *bias* and *variance*.

Bias refers to the gap between your predicted value and the actual value. In the case of high bias, your predictions are likely to be skewed in a certain direction away from the actual values. Variance describes how scattered your predicted values are. Bias and variance can be best understood by analyzing the following visual representation.

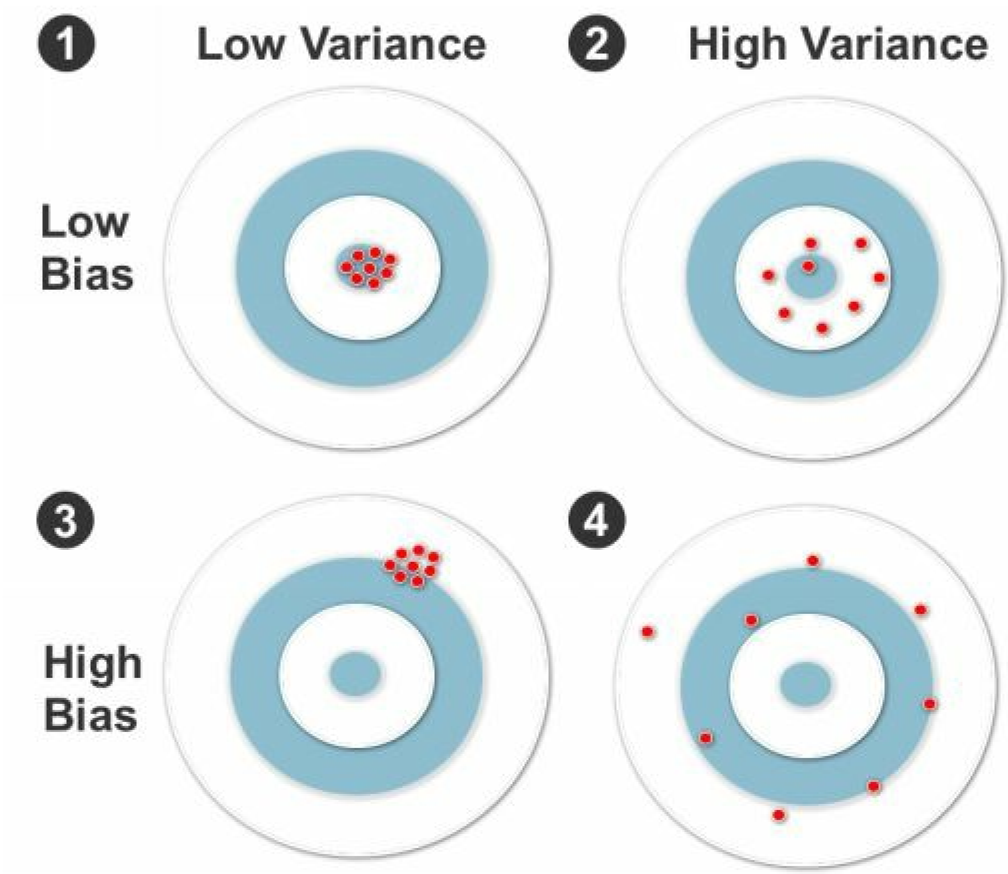


Figure 2: Shooting targets used to represent bias and variance

Shooting targets, as seen in Figure 2, are not a visual chart used in machine learning, but it does help to explain bias and variance. Imagine that the center of the target, or the bull's-eye, perfectly predicts the correct value of your model. The dots marked on the target then represent an individual realization of your model based on your training data. In certain cases, the dots will be densely positioned close to the bull's-eye, ensuring that predictions made by the model are close to the actual data. In other cases, the training data will be scattered across the target. The more the dots deviate from the bull's-eye, the higher the bias and the less accurate the model will be in its overall predictive ability.

In the first target, we can see an example of low bias and low variance. Bias is low because the hits are closely aligned to the center and there is low variance because the hits are densely positioned in one location.

The second target (located on the right of the first row) shows a case of low bias and high variance. Although the hits are not as close to the bull's-eye as the previous example, they are still near to the center and bias is therefore relatively low. However, there is high variance this time because the hits are

spread out from each other.

The third target (located on the left of the second row) represents high bias and low variance and the fourth target (located on the right of the second row) shows high bias and high variance.

Ideally, you want a situation where there is low variance and low bias. In reality, though, there is more often a trade-off between optimal bias and variance. Bias and variance both contribute to error, but it is the prediction error that you want to minimize, not bias or variance specifically.

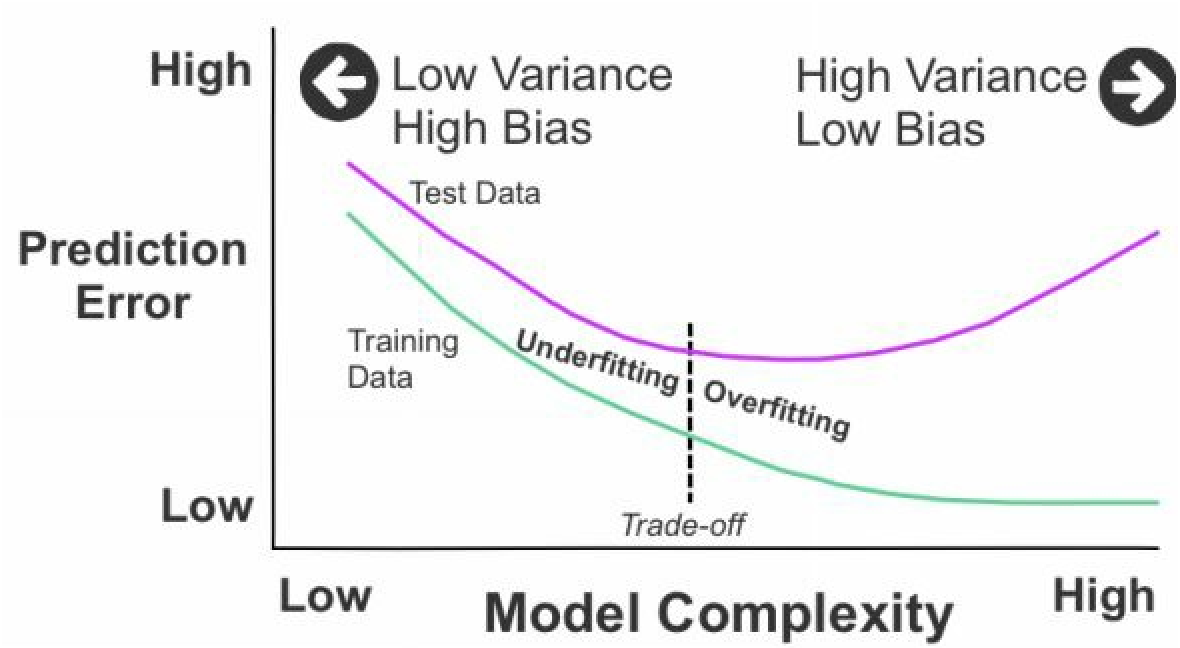


Figure 3: Model complexity based on prediction error

In Figure 3, we can see two lines moving from left to right. The line above represents the test data and the line below represents the training data. From the left, both lines begin at a point of high prediction error due to low variance and high bias. As they move from left to right they change to the opposite: high variance and low bias. This leads to low prediction error in the case of the training data and high prediction error for the test data. In the middle of the chart is an optimal balance of prediction error between the training and test data. This is a common case of bias-variance trade-off.

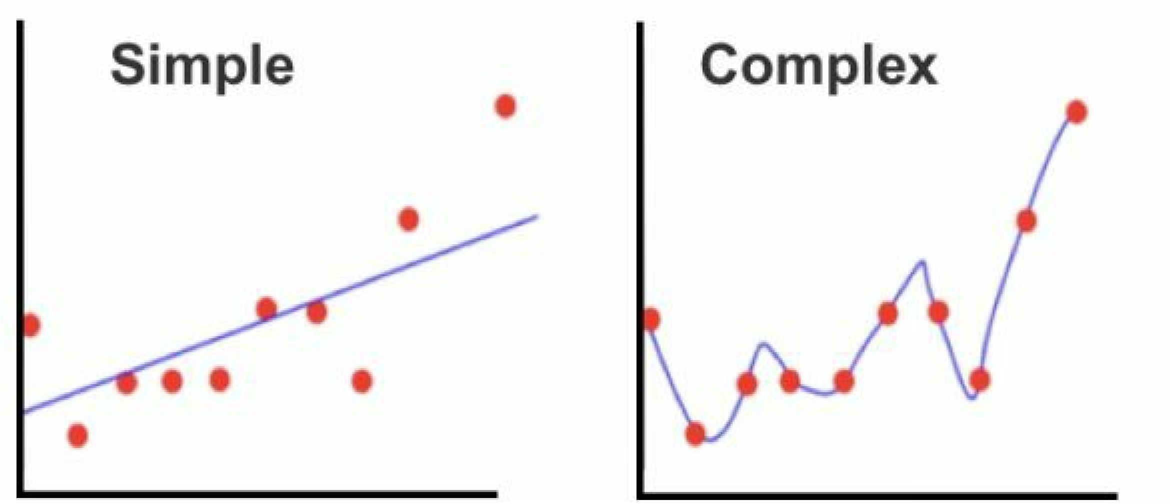


Figure 4: Underfitting on the left and overfitting on the right

Mismanaging the bias-variance trade-off can lead to poor results. As seen in Figure 4, this can result in the model becoming overly simple and inflexible (underfitting) or overly complex and flexible (overfitting).

Underfitting (low variance, high bias) on the left and overfitting (high variance, low bias) on the right are shown in these two scatterplots. A natural temptation is to add complexity to the model (as shown on the right) in order to improve accuracy, but which can, in turn, lead to overfitting. An overfitted model will yield accurate predictions from the training data but prove less accurate at formulating predictions from the test data. Overfitting can also occur if the training and test data aren't randomized before they are split and patterns in the data aren't distributed across the two segments of data.

Underfitting is when your model is overly simple, and again, has not scratched the surface of the underlying patterns in the dataset. Underfitting can lead to inaccurate predictions for both the training data and test data. Common causes of underfitting include insufficient training data to adequately cover all possible combinations, and situations where the training and test data were not properly randomized.

To eradicate both underfitting and overfitting, you may need to modify the model's hyperparameters to ensure that they fit patterns in both the training and test data and not just one-half of the data. A suitable fit should acknowledge major trends in the data and play down or even omit minor variations. This may also mean re-randomizing the training and test data or adding new data points so as to better detect underlying patterns. However, in most instances, you will probably need to consider switching algorithms or modifying your hyperparameters based on trial and error to minimize and

manage the issue of bias-variance trade-off.

Specifically, this might entail switching from linear regression to non-linear regression to reduce bias by increasing variance. Or it could mean increasing "*k*" in *k*-NN to reduce variance (by averaging together more neighbors). A third example could be reducing variance by switching from a single decision tree (which is prone to overfitting) to a random forest with many decision trees.

Another effective strategy to combat overfitting and underfitting is to introduce *regularization*. Regularization artificially amplifies bias error by penalizing an increase in a model's complexity. In effect, this add-on parameter provides a warning alert to keep high variance in check while the original parameters are being optimized.

Another effective technique to contain overfitting and underfitting in your model is to perform cross validation, as covered earlier in Chapter 6, to minimize any discrepancies between the training data and the test data.

10

ARTIFICIAL NEURAL NETWORKS

This penultimate chapter on machine learning algorithms brings us to artificial neural networks (ANN) and the gateway to reinforcement learning. Artificial neural networks, also known as neural networks, is a popular machine learning technique to process data through layers of analysis. The naming of artificial neural networks was inspired by the algorithm's resemblance to the human brain.

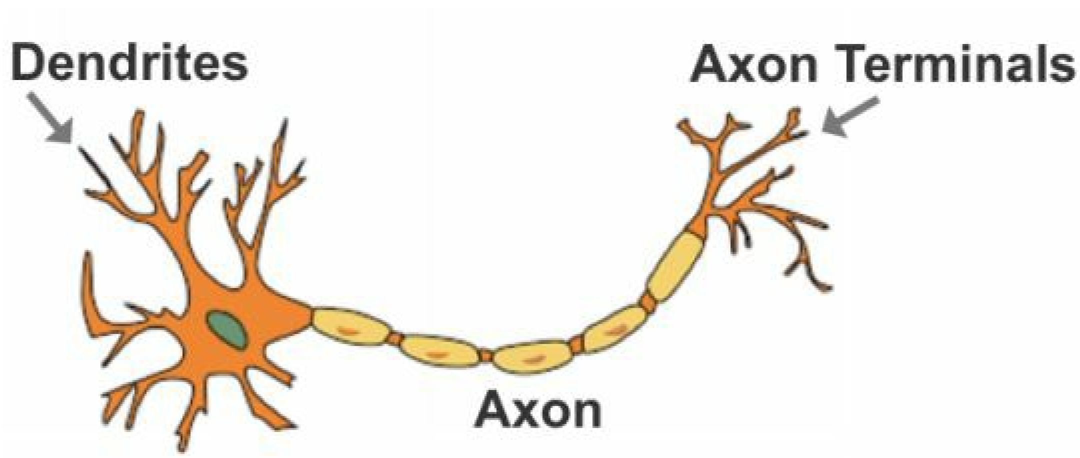


Figure 1: Anatomy of a human neuron

The human brain contains interconnected neurons with dendrites that receive inputs. From these inputs, the neuron produces an electric signal output from the axon and then emits these signals through axon terminals to other neurons.

Similar to neurons in the human brain, artificial neural networks are formed by interconnected neurons, also called *nodes,* which interact with each other through axons, called *edges*. In a neural network, the nodes are stacked up in layers and generally start with a broad base. The first layer consists of raw data such as numeric values, text, images or sound, which are divided into nodes. Each node then sends information to the next layer of nodes through the network's edges.

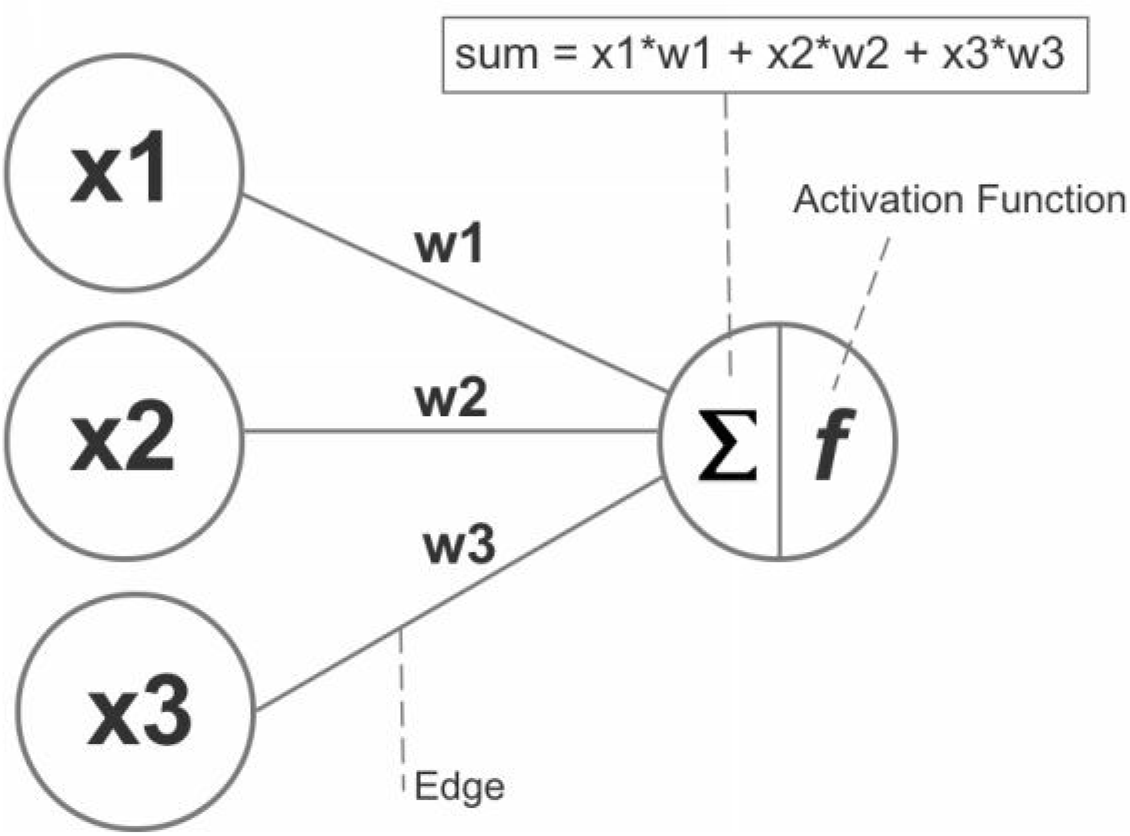


Figure 2: The nodes, edges/weights, and sum/activation function of a basic neural network

Each edge has a numeric weight (algorithm) that can be altered and formulated based on experience. If the sum of the connected edges satisfies a set threshold, known as the activation function, it will activate a neuron at the next layer. However, if the sum of the connected edges does not meet the set threshold, the activation will not be triggered. This results in an *all or nothing* arrangement.

Note, also, that the weights along each edge are unique to ensure that the nodes fire differently (as seen in Figure 3) and they don't all return the same outcome.

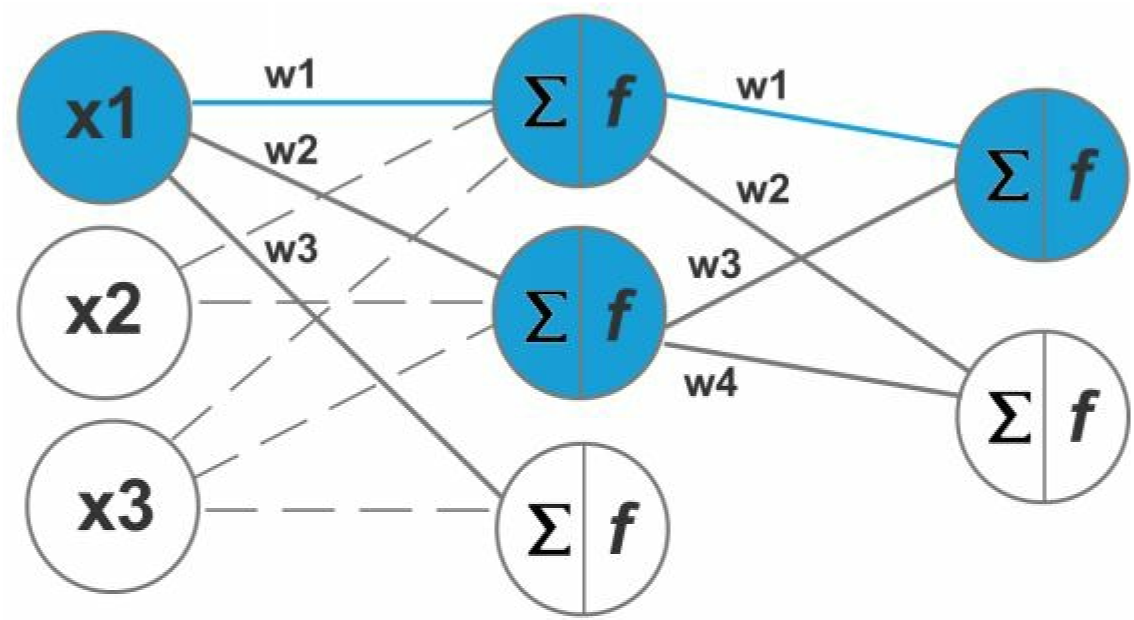


Figure 3: Unique edges to produce different outcomes

To train the network through supervised learning, the model's predicted output is compared to the actual output (that is known to be correct) and the difference between these two results is measured and is known as the *cost* or *cost value*. The purpose of training is to reduce the cost value until the model's prediction closely matches the correct output. This is achieved by incrementally tweaking the network's weights until the lowest possible cost value is obtained. This process of training the neural network is called *back-propagation*. Rather than navigate left to right like how data is fed into a neural network, back-propagation is done in reverse and runs from the output layer from the right towards the input layer on the left.

One of the downsides of neural networks is that they operate as a black-box in the sense that while the network can approximate accurate outcomes, tracing its structure reveals limited or no insight on the variables that impact the outcome. For example, when using a neural network to predict the probable outcome of a Kickstarter (the world's largest funding platform for creative projects) campaign, the network will analyze a number of variables such as campaign category, currency, deadline, and minimum pledge amount, but it won't be able to specify their relationships to the final outcome. Moreover, it's possible for two neural networks with a different topology and different weights to produce the same output, which makes it even more difficult to trace variable relationships to the output. Examples of non-black-box models are regression techniques and decision trees.

So, when should you use a back-box neural network? In general, neural networks are best for solving problems with highly complex patterns and especially those that are difficult for computers to solve but simple and almost trivial for humans. An obvious example is a CAPTCHA (Completely Automated Public Turing test to tell Computers and Humans Apart) challenge-response test that is used on websites to determine whether an online user is an actual human. There are numerous blog posts online that demonstrate how you can crack a CAPTCHA test using neural networks. Another example is identifying whether a pedestrian will step in the path of an oncoming vehicle as used in self-driving vehicles to avoid the case of an accident.

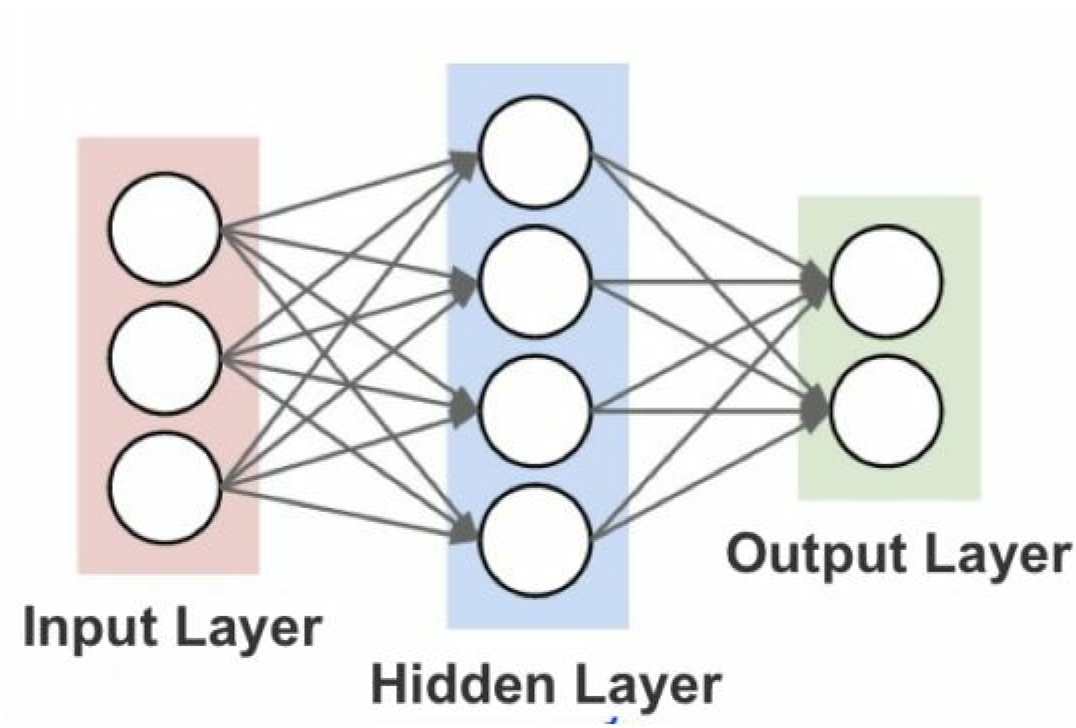


Figure 4: The three general layers of a neural network

A typical neural network can be divided into input, hidden, and output layers. Data is first received by the input layer, where broad features are detected. The hidden layer(s) then analyze and process the data. Based on previous computations, the data becomes streamlined through the passing of each hidden layer. The final result is shown as the output layer.

The middle layers are considered hidden layers because, like human vision, they covertly break down objects between the input and output layers. For example, when humans see four lines connected in the shape of a square we

instantly recognize those four lines as a square. We don't notice the lines as four independent lines with no relationship to each other. Our brain is conscious only of the output layer. Neural networks work much the same way in that they break down data into layers and examine the hidden layers to produce a final output.

While there are many techniques to assemble the nodes of a neural network, the simplest method is the feed-forward network. In a feed-forward network, signals flow only in one direction and there is no loop in the network.

The most basic form of a feed-forward neural network is the *perceptron*.

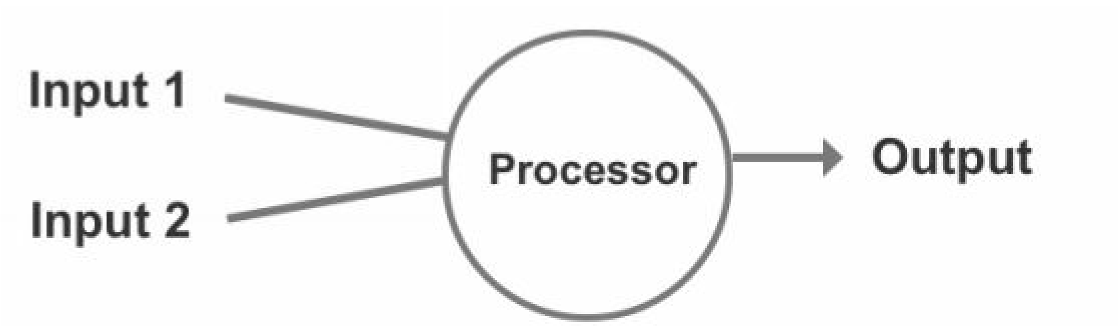


Figure 5: Visual representation of a perceptron neural network

A perceptron consists of one or more inputs, a processor, and a single output. Within a perceptron model, inputs:

1) Are fed into the processor (neuron)
2) Are processed
3) Generate output

As an example, let's say we have a perceptron consisting of two inputs:

Input 1: 3x = 24

Input 2: 2x = 16

We then add a random weight to these two inputs and they are sent into the neuron to be processed.

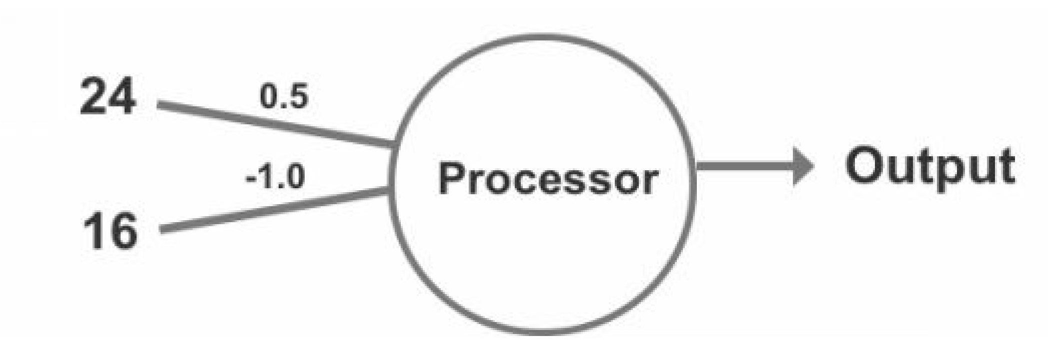


Figure 6: Weights are added to the perceptron

Weights
Input 1: 0.5
Input 2: -1.0
Next, multiply each weight by its input:
Input 1: 24 * 0.5 = 12
Input 2: 16 * -1.0 = -16
Passing the sum of the edge weights through the activation function generates the perceptron's output.
A key feature of the perceptron is that it only registers two possible outcomes, "1" and "0." The value of "1" triggers the activation function and the value of "0" does not. Although the perceptron is binary in nature (1 or 0), there are various ways in which we can configure the activation function. In this example, we made the activation function ≥0. This means that if the sum is a positive number or zero, the output is 1. If the sum is a negative number, the output is 0.

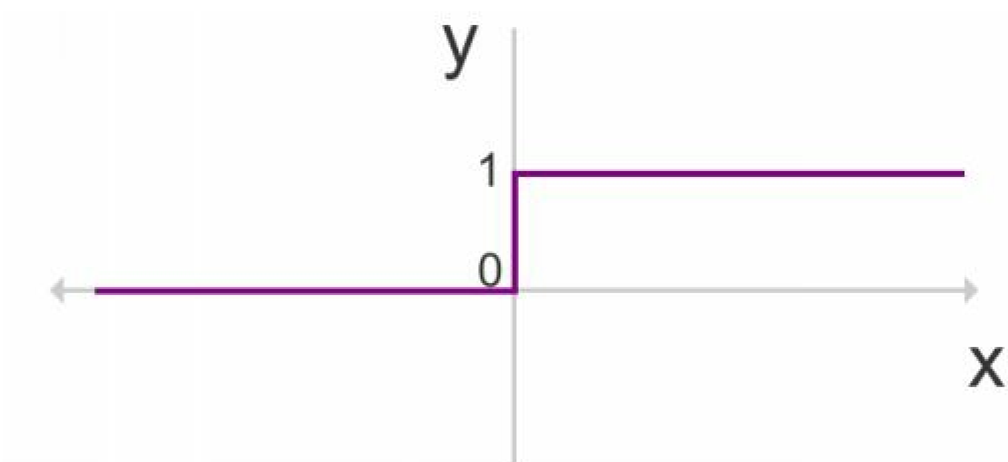


Figure 7: Activation function where the output (y) is 0 when x is negative, and the output (y) is 1 when x is positive

Thus:
Input 1: 24 * 0.5 = 12
Input 2: 16 * -1.0 = -16
Sum (Σ): 12 + -16 = - 4
As a numeric value less than zero, our result will register as "0" and therefore not trigger the activation function of the perceptron.
However, we can also modify the activation threshold to a completely

different rule, such as:

x > 3, y = 1

x ≤ 3, y = 0

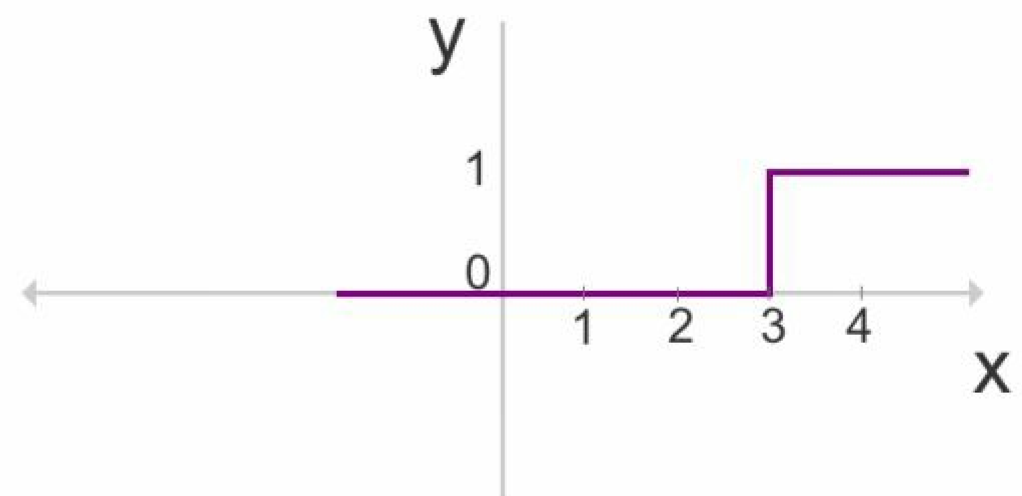


Figure 8: Activation function where the output (y) is 0 when x is equal or less than 3, and the output (y) is 1 when x is greater than 3

When working with a larger model of neural network layers, a value of "1" will be configured to pass the output to the next layer. Conversely, a "0" value is configured to be ignored and will not be passed to the next layer for processing.

In supervised learning, perceptrons can be used to train data and develop a prediction model. The steps to training data are as follows:

1) Inputs are fed into the processor (neurons/nodes).
2) The perceptron estimates the value of those inputs.
3) The perceptron computes the error between the estimate and the actual value.
4) The perceptron adjusts its weights according to the error.
5) Repeat the previous four steps until you are satisfied with the model's accuracy. The training model can then be applied to the test data.

The weakness of a perceptron is that, because the output is binary (1 or 0), small changes in the weights or bias in any single perceptron within a larger neural network can induce polarizing results. This can lead to dramatic changes within the network and a complete flip in regards to the final output. As a result, this makes it very difficult to train an accurate model that can be successfully applied to test data and future data inputs.

An alternative to the perceptron is the *sigmoid neuron*. A sigmoid neuron is

very similar to a perceptron, but the presence of a sigmoid function rather than a binary model now accepts any value between 0 and 1. This enables more flexibility to absorb small changes in edge weights without triggering inverse results—as the output is no longer binary. In other words, the output result won't flip just because of one minor change to an edge weight or input value.

$$y = \frac{1}{1+e^{-x}}$$

Figure 9: The sigmoid equation, as first seen in logistic regression

While more flexible than a perceptron, a sigmoid neuron cannot generate negative values. Hence, a third option is the *hyperbolic tangent function*.

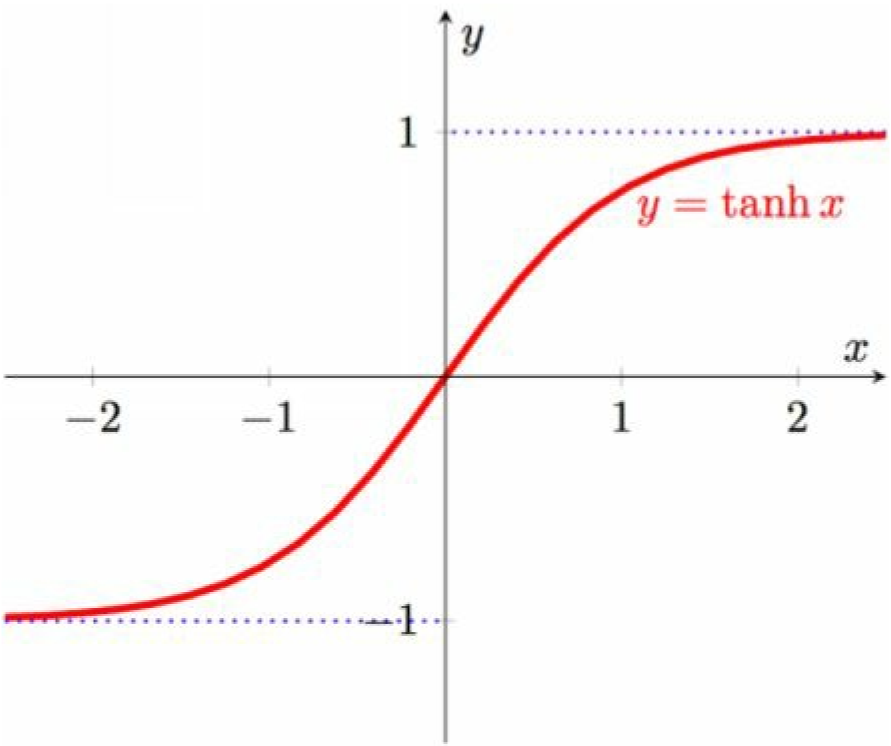


Figure 10: A hyperbolic tangent function graph

We have so far discussed basic neural networks; to create a more advanced neural network, we can link sigmoid neurons and other classifiers to create a network with a higher number of layers or combine multiple perceptrons to form a multi-layer perceptron.

For analyzing simple patterns, a basic neural network or an alternative classification tool such as logistic regression and *k*-nearest neighbors is

generally sufficient for the purpose of analysis. However, as the patterns in the data become more complicated—especially in the form of a high number of inputs such as the total number of pixels in an image—a basic or shallow model is no longer reliable or capable of analysis. This is because the model becomes exponentially complex as the number of inputs rises and in the case of neural networks this means more layers to manage more input nodes. A neural network, with a deep number of layers, however, is able to break down complex patterns into simpler patterns as demonstrated in Figure 11.

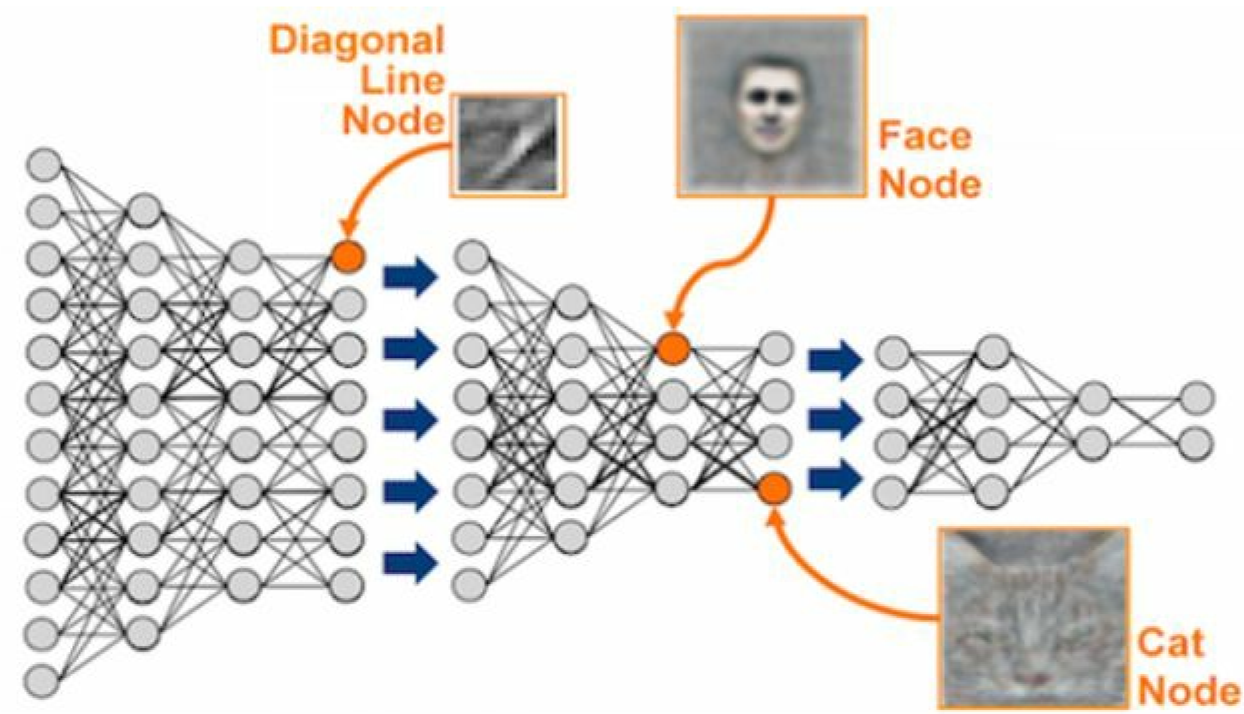


Figure 11: Facial recognition using deep learning. *Source: kdnuggets.com*

This deep network uses edges to detect different physical features to recognize faces, such as a diagonal line. Like building blocks, the network combines the node results to classify the input as, say, a human's face or a cat's face and then processes that further to recognize a specific individual's face.

This is known as deep learning. What makes deep learning "deep" is the stacking of at least 5-10 node layers, with advanced object recognition using upwards of 150 layers.

Object recognition, as used by self-driving vehicles to recognize objects such as pedestrians and other vehicles, is a popular application of deep learning today. Other common applications of deep learning include time series

analysis to analyze data trends measured over particular time periods or intervals, speech recognition, and text processing tasks including sentiment analysis, topic segmentation, and named entity recognition. More usage scenarios and commonly paired deep learning techniques are listed in Figure 12.

	Recurrent Network	Recursive Neural Tensor Network	Deep Belief Network	Convolution Network	MLP
Text Processing	✔	✔		✔	
Image Recognition			✔	✔	
Object Recognition		✔		✔	
Speech Recognition	✔				
Time Series Analysis	✔				
Classification			✔	✔	✔

Figure 12: Common usage scenarios and paired deep learning techniques

As can be seen from the table, multi-layer perceptrons have been largely superseded by new deep learning techniques such as convolution networks, recurrent networks, deep belief networks, and recursive neural tensor networks (RNTN). These more advanced iterations of a neural network can be used effectively across a number of practical applications that are currently in vogue today. Although convolution networks are arguably the most popular and powerful of deep learning techniques, new methods and variations are continuously evolving.

11

DECISION TREES

The fact that neural networks can be applied to a broader range of machine learning problems than any other technique has led some pundits to hail neural networks as the ultimate machine learning algorithm. However, this is not to say that neural networks fit the bill as a statistical silver bullet. In various cases, neural networks fall short and decision trees are held up as a popular counterargument.

The massive reserve of data and computational resources that neural networks demand is one obvious pitfall. Only after training on millions of tagged examples can Google's image recognition engine reliably recognize classes of simple objects (such as dogs). But how many dog pictures do you need to show to the average four-year-old before they “get it?”

Decision trees, on the other hand, provide high-level efficiency and easy interpretation. These two benefits make this simple algorithm popular in the space of machine learning.

As a supervised learning technique, decision trees are used primarily for solving classification problems, but they can be applied to solve regression problems too.

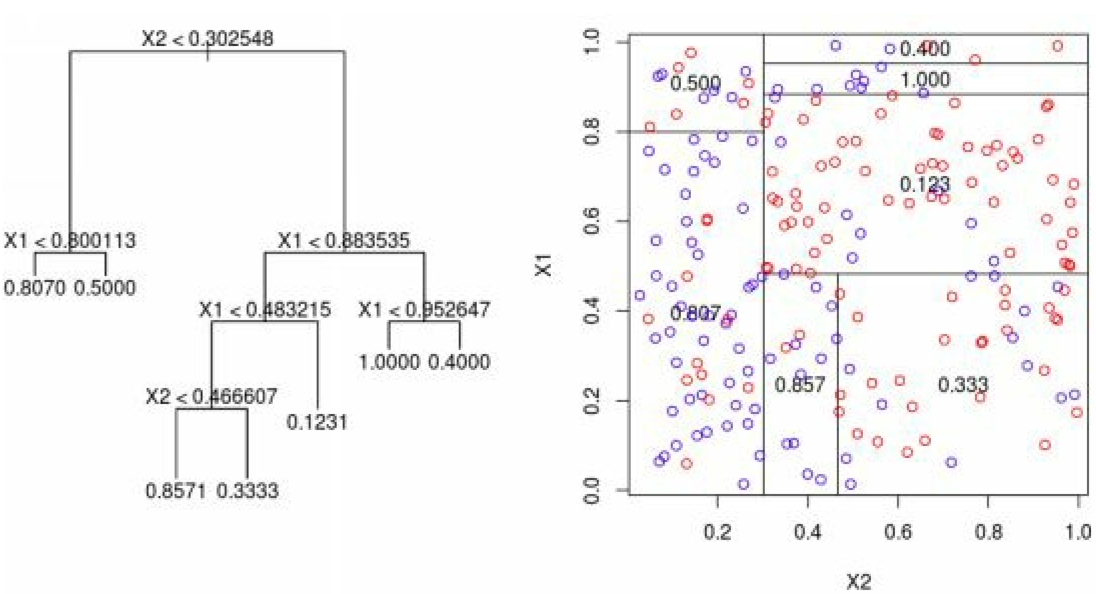


Figure 1: Example of a regression tree. *Source: http://freakonometrics.hypotheses.org/*

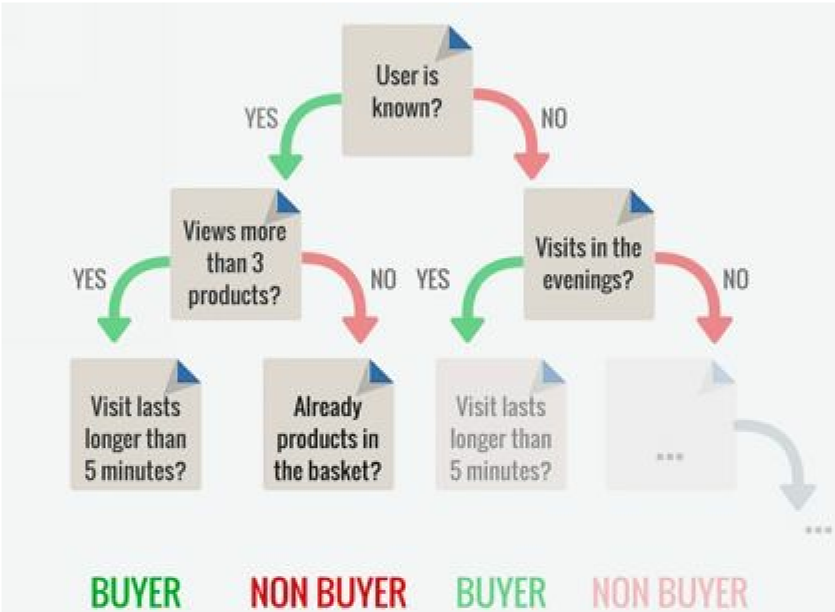


Figure 2: Example of a classification tree. *Source: http://blog.akanoo.com*

Classification trees can use quantitative and categorical data to model categorical outcomes. Regression trees also use quantitative and categorical data but instead model quantitative outcomes.

Decision trees start with a root node, which acts as a starting point (at the top), and is followed by splits that produce branches. The statistical/mathematical term for these branches is *edges*. The branches then link to leaves, known also as nodes, which form decision points. A final categorization is produced when a leaf does not generate any new branches and results in what is known as a terminal node.

Decision trees thus not only break down and explain how classification or regression is formulated, but they also produce a neat visual flowchart you can show to others. The ease of interpretation is a strong advantage of using decision trees, and they can be applied to a wide range of use cases.

Real-life examples include picking a scholarship recipient, assessing an applicant for a home loan, predicting e-commerce sales, or selecting the right job applicant. When a customer or applicant queries why they weren't selected for a particular scholarship, home loan, job, etc., you can pass them the decision tree and let them see the decision-making process for themselves.

Building a Decision Tree

Decision trees are built by first splitting data into two groups. This binary splitting process is then repeated at each branch (layer). The aim is to select a binary question that best splits the data into two homogenous groups at each branch of the tree, such that it minimizes the level of data entropy at the next.

Entropy is a mathematical term that explains the measure of variance in the data among different classes. In simple terms, we want the data at each layer to be more homogenous than at the last.

We thus want to pick a "greedy" algorithm that can reduce the level of entropy at each layer of the tree. One such greedy algorithm is the Iterative Dichotomizer (ID3), invented by J.R. Quinlan. This is one of three decision tree implementations developed by Quinlan, hence the "3."

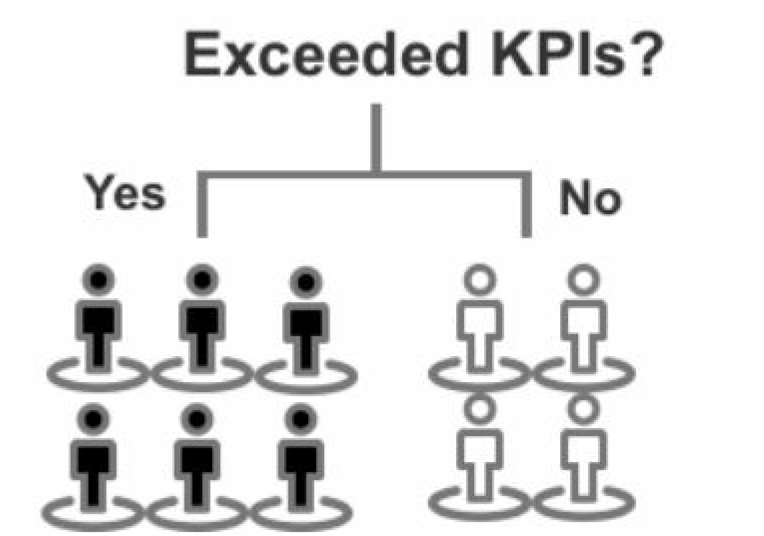
ID3 applies entropy to determine which binary question to ask at each layer of the decision tree. At each layer, ID3 identifies a variable (converted into a binary question) that will produce the least entropy at the next layer. Let's consider the following example to better understand how this works.

Employees	Exceeded KPIs	Leadership Capability	Aged < 30	Outcome
6	6	2	3	Promoted
4	0	2	4	Not promoted

Variable 1 (exceeded Key Performance Indicators) produces:

- Six promoted employees who exceeded their KPIs (Yes)
- Four employees who didn't exceed their KPIs and who were not promoted (No)

This variable produces two homogenous groups at the next layer of the decision tree.

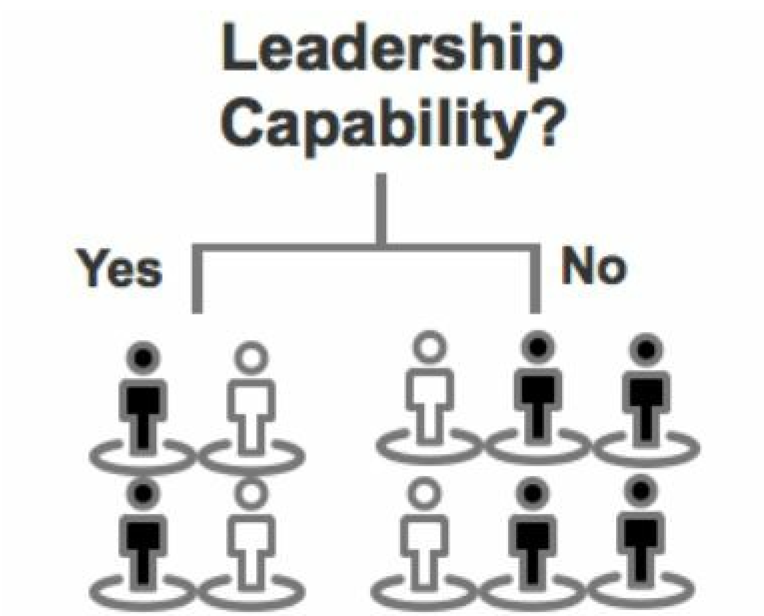


Black = Promoted, White = Not Promoted

Variable 2 (leadership capability) produces:

- Two promoted employees with leadership capabilities (Yes)
- Four promoted employees with no leadership capabilities (No)
- Two employees with leadership capabilities who were not promoted (Yes)
- Two employees with no leadership capabilities who were not promoted (No)

This variable produces two groups of mixed data points.

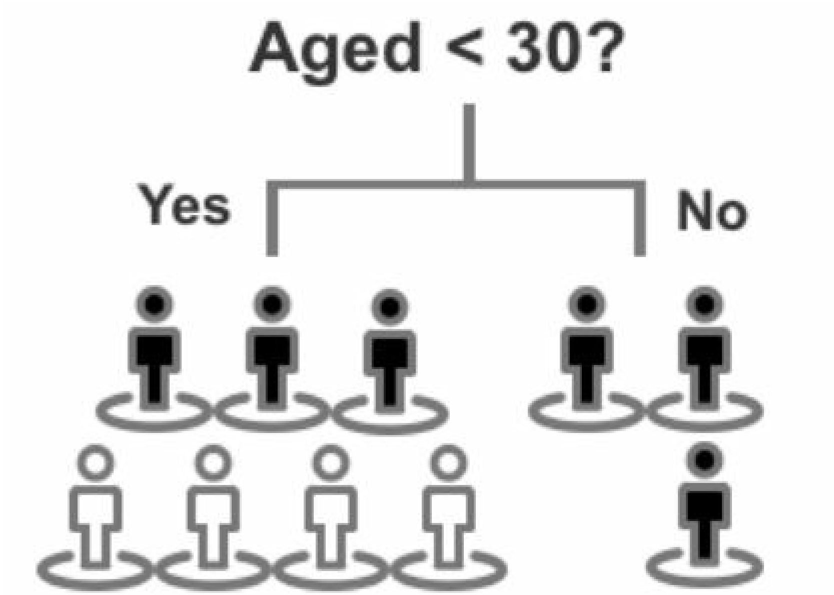


Black = Promoted, White = Not Promoted

Variable 3 (aged under thirty) produces:

- Three promoted employees aged under thirty (Yes)
- Three promoted employees aged over thirty (No)
- Four employees aged under thirty who were not promoted (Yes)

This variable produces one homogenous group and one mixed group of data points.



Black = Promoted, White = Not Promoted

Of these three variables, variable 1 (Exceeded KPIs) produces the best result with two perfectly homogenous groups. Variable 3 produces the second best result, as one leaf is homogenous. Variable 2 produces two leaves that are not homogenous. Variable 1 would therefore be selected as the first binary question to split this dataset.

Whether it is ID3 or another algorithm, this process of splitting data into binary partitions, known as *recursive partitioning*, is repeated until a stopping criterion is met. This stopping point could be based on a range of criteria, such as:

- When all leaves contain less than 3-5 items
- When a branch produces a result that places all items in one binary leaf

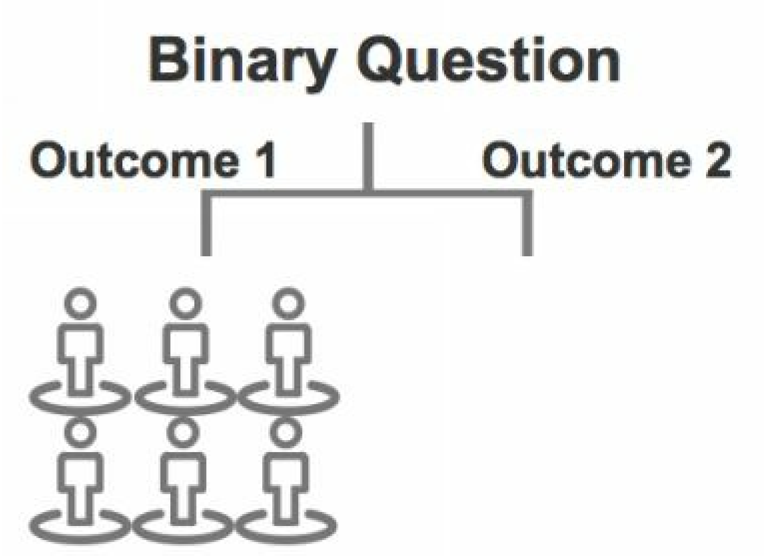


Figure 3: Example of a stopping criteria

A caveat to remember when using decision trees is their susceptibility to overfitting. The cause of overfitting, in this case, is the training data. Taking into account the patterns that exist in your training data, a decision tree is precise at training the first round of data. However, the same decision tree may then fail to predict the test data, as there could be rules that it is yet to encounter or because the training or test data were not representative of the entire dataset. Moreover, because decision trees are formed from repeatedly splitting data points into two partitions, a slight change in how the data is split at the top or middle of the tree can dramatically alter the final prediction. This can produce a different tree altogether! The offender, in this case, is our greedy algorithm.

From the very first split of the data, the greedy algorithm fixes its attention on picking a binary question that best partitions data into two homogenous groups. Like a boy sitting in front of a box of cupcakes, the greedy algorithm is oblivious to the future repercussions of its short-term actions. The binary question it uses to initially split the data does not guarantee the most accurate final prediction. Rather, a less effective initial split may produce a more accurate outcome.

In sum, decision trees are highly visual and effective at classifying a single set of data, but they can be inflexible and vulnerable to overfitting.

Random Forests

Rather than striving for the most efficient split at each round of recursive partitioning, an alternative technique is to construct multiple trees and

combine their predictions to select an optimal path of classification or prediction. This involves a randomized selection of binary questions to grow multiple different decision trees, known as *random forests*. In the industry, you will also often hear people refer to this process as “bootstrap aggregating” or “bagging.”

Bootstrap Aggregating

Figure 4: “Bagging” is a creative abbreviation of “Bootstrap Aggregating”

The key to understanding random forests is to first understand bootstrap sampling. There’s little use in compiling five or ten identical models—there needs to be some element of variation. This is why bootstrap sampling draws on the same dataset but extracts a different variation of the data at each turn.

Hence, in growing random forests, multiple varying copies of the training data are first run through each of the trees. For classification problems, bagging undergoes a process of voting to generate the *final class*. The results from each tree are compared and voted on to create an optimal tree to produce the final model, known as the final class. For regression problems, value averaging is used to generate a final prediction.

Bootstrapping is also sometimes called weakly-supervised (you will recall we explored supervised and unsupervised learning in Chapter 3) because it trains classifiers using a random subset of features and fewer variables than those actually available.

Boosting

Another variant of multiple decision trees is the popular technique of *boosting,* which are a family of algorithms that convert “weak learners” to “strong learners.” The underlying principle of boosting is to add weights to iterations that were misclassified in earlier rounds. This can be interpreted as similar to a language teacher offering after-school tutoring to the weakest students in the class in order to improve the average test results of the entire class.

A popular boosting algorithm is gradient boosting. Rather than selecting combinations of binary questions at random (like random forests), gradient boosting selects binary questions that improve prediction accuracy for each new tree. Decision trees are therefore grown sequentially, as each tree is created using information derived from the previous decision tree.

The way this works is that mistakes incurred with the training data are recorded and then applied to the next round of training data. At each iteration, weights are added to the training data based on the results of the previous iteration. Higher weighting is applied to instances that were incorrectly predicted from the training data, and instances that were correctly predicted receive less weighting. The training and test data are then compared and errors are again logged in order to inform weighting at each subsequent round. Earlier iterations that do not perform well, and that perhaps misclassified data, can thus be improved upon through further iterations. This process is repeated until there is a low level of error. The final result is then obtained from a weighted average of the total predictions derived from each model.

While this approach mitigates the issue of overfitting, it does so with fewer trees than the bagging approach. In general, the more trees you add to a random forest, the greater its ability to thwart overfitting. Conversely, with gradient boosting, too many trees may cause overfitting and caution should be taken as new trees are added.

One drawback of using random forests and gradient boosting is that we return to a black-box technique and sacrifice the visual simplicity and ease of interpretation that comes with a single decision tree.

12

ENSEMBLE MODELING

One of the most effective machine learning methodologies is *ensemble modeling*, also known as *ensembles*. Ensemble modeling combines statistical techniques to create a model that produces a unified prediction. It is through combining estimates and following the wisdom of the crowd that ensemble modeling performs a final classification or outcome with better predictive performance. Naturally, ensemble models are a popular choice when it comes to machine learning competitions like the Netflix Competition and Kaggle competitions.

Ensemble models can be classified into various categories including sequential, parallel, homogenous, and heterogeneous. Let's start by first looking at sequential and parallel models. For sequential ensemble models, prediction error is reduced by adding weights to classifiers that previously misclassified data. Gradient boosting and AdaBoost are two examples of sequential models. Conversely, parallel ensemble models work concurrently and reduce error by averaging. Decision trees are an example of this technique.

Ensemble models can also be generated using a single technique with numerous variations (known as a homogeneous ensemble) or through different techniques (known as a heterogeneous ensemble). An example of a homogeneous ensemble model would be numerous decision trees working together to form a single prediction (bagging). Meanwhile, an example of a heterogeneous ensemble would be the usage of *k*-means clustering or a neural network in collaboration with a decision tree model.

Naturally, it is important to select techniques that complement each other. Neural networks, for instance, require complete data for analysis, whereas decision trees can effectively handle missing values. Together, these two techniques provide added value over a homogeneous model. The neural network accurately predicts the majority of instances that provide a value and the decision tree ensures that there are no "null" results that would otherwise be incurred from missing values in a neural network. The other advantage of ensemble modeling is that aggregated estimates are generally more accurate

than any single estimate.

There are various subcategories of ensemble modeling; we have already touched on two of these in the previous chapter. Four popular subcategories of ensemble modeling are bagging, boosting, a bucket of models, and stacking.

Bagging, as we know, is short for "boosted aggregating" and is an example of a homogenous ensemble. This method draws upon randomly drawn datasets and combines predictions to design a unified model based on a voting process among the training data. Expressed in another way, bagging is a special process of model averaging. Random forest, as we know, is a popular example of bagging.

Boosting is a popular alternative technique that addresses error and data misclassified by the previous iteration to form a final model. Gradient boosting and AdaBoost are both popular examples of boosting.

A **bucket of models** trains numerous different algorithmic models using the same training data and then picks the one that performed most accurately on the test data.

Stacking runs multiple models simultaneously on the data and combines those results to produce a final model. This technique is currently very popular in machine learning competitions, including the Netflix Prize. (Held between 2006 and 2009, Netflix offered a prize for a machine learning model that could improve their recommender system in order to produce more effective movie recommendations. One of the winning techniques adopted a form of linear stacking that combined predictions from multiple predictive models.)

Although ensemble models typically produce more accurate predictions, one drawback to this methodology is, in fact, the level of sophistication. Ensembles face the same trade-off between accuracy and simplicity as a single decision tree versus a random forest. The transparency and simplicity of a simple technique, such as a decision tree or *k*-nearest neighbors, is lost and instantly mutated into a statistical black-box. Performance of the model will win out in most cases, but the transparency of your model is another factor to consider when determining your preferred methodology.

13

BUILDING A MODEL IN PYTHON

After examining the statistical underpinnings of numerous algorithms, it's time to turn our attention to building an actual machine learning model. Although there are various options in regards to programming languages (as outlined in Chapter 4), for this exercise we will use Python because it is quick to learn and it's an effective programming language for anyone interested in manipulating and working with large datasets.

If you don't have any experience in programming or programming with Python, there's no need to worry. The key purpose of this chapter is to understand the methodology and steps behind building a basic machine learning model.

In this exercise, we will design a house price valuation system using gradient boosting by following these six steps:

1) Set up the development environment
2) Import the dataset
3) Scrub the dataset
4) Split the data into training and test data
5) Select an algorithm and configure its hyperparameters
6) Evaluate the results

1) Set up the development environment

The first step is to prepare our development environment. For this exercise, we will be working in Jupyter Notebook, which is an open-source web application that allows editing and sharing of notebooks.

You can download Jupyter Notebook from: http://jupyter.org/install.html

Jupyter Notebook can be installed using the Anaconda Distribution or Python's package manager, pip. There are instructions available on the Jupyter Notebook website that outline both options. As an experienced Python user, you may wish to install Jupyter Notebook via pip. For beginners, I recommend selecting the Anaconda Distribution option, which offers an easy click-and-drag setup.

This particular installation option will direct you to the Anaconda website. From there, you can select your preferred installation for Windows, macOS, or Linux. Again, you can find instructions available on the Anaconda website according to your choice of operating system.
After installing Anaconda to your machine, you will have access to a number of data science applications including rstudio, Jupyter Notebook, and graphviz for data visualization. For this exercise, you will need to select Jupyter Notebook by clicking on "Launch" inside the Jupyter Notebook tab.

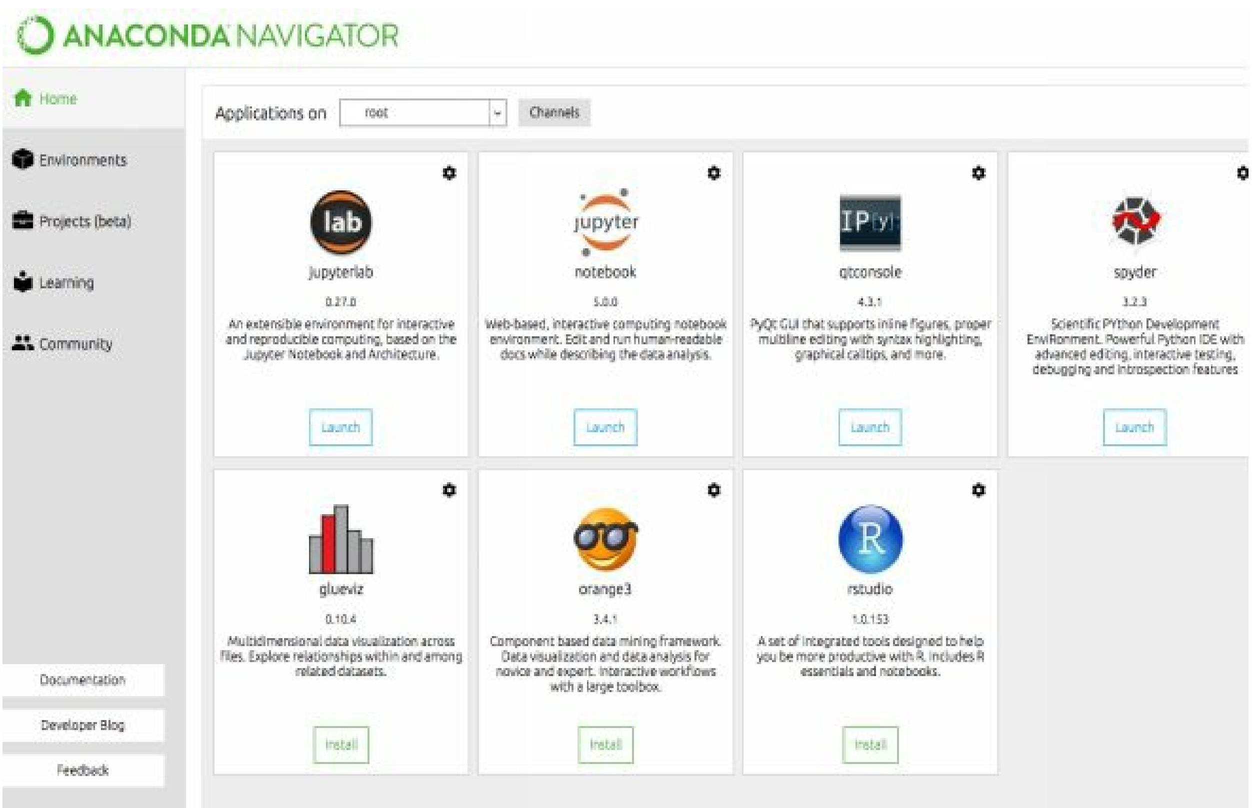


Figure 1: The Anaconda Navigator portal

To initiate Jupyter Notebook, run the following command from the Terminal (for Mac/Linux) or Command Prompt (for Windows):

```
jupyter notebook
```

Terminal/Command Prompt will then generate a URL for you to copy and paste into your web browser. Example: http://localhost:8888/
Copy and paste the generated URL into your web browser to load Jupyter Notebook. Once you have Jupyter Notebook open in your browser, click on

“New” in the top right-hand corner of the web application to create a new “Notepad” project, and then select “Python 3.”

The final step is to install the necessary libraries required to complete this exercise. You will need to install Pandas and a number of libraries from Scikit-learn into the notepad.

In machine learning, each project will vary in regards to the libraries required for import. For this particular exercise, we are using gradient boosting (ensemble modeling) and mean absolute error to measure performance.

You will need to import each of the following libraries and functions by entering these exact commands in Jupyter Notebook:

```
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn import ensemble
from sklearn.metrics import mean_absolute_error
from sklearn.externals import joblib
```

Don’t worry if you don’t recognize each of the imported libraries in the code snippet above. These libraries will be referred to in later steps.

2) Import the dataset

The next step is to import the dataset. For this exercise, I have selected a free and publicly available dataset from kaggle.com which contains house, unit, and townhouse prices in Melbourne, Australia. This dataset comprises data scraped from publicly available listings posted weekly on www.domain.com.au. The dataset contains 14,242 property listings and 21 variables including address, suburb, land size, number of rooms, price, longitude, latitude, postcode, etc.

Please note that the property values in this dataset are expressed in Australian Dollars—$1 AUD is approximately $0.77 USD (as of 2017).

Download the Melbourne Housing Market dataset from this link: https://www.kaggle.com/anthonypino/melbourne-housing-market

After registering a free account and logging into kaggle.com, download the dataset as a zip file. Next, unzip the downloaded file and import into Jupyter Notebook. To import the dataset, you can utilize the read_csv function to load the data into a Pandas dataframe.

```
df = pd.read_csv('~/Downloads/Melbourne_housing_FULL-26-09-2017.csv')
```

This command will directly import the dataset. However, please note that the exact file path will depend on the saved location of your dataset. For example, if you saved the CSV file to your desktop, you would need to read in the .csv file using the following command:

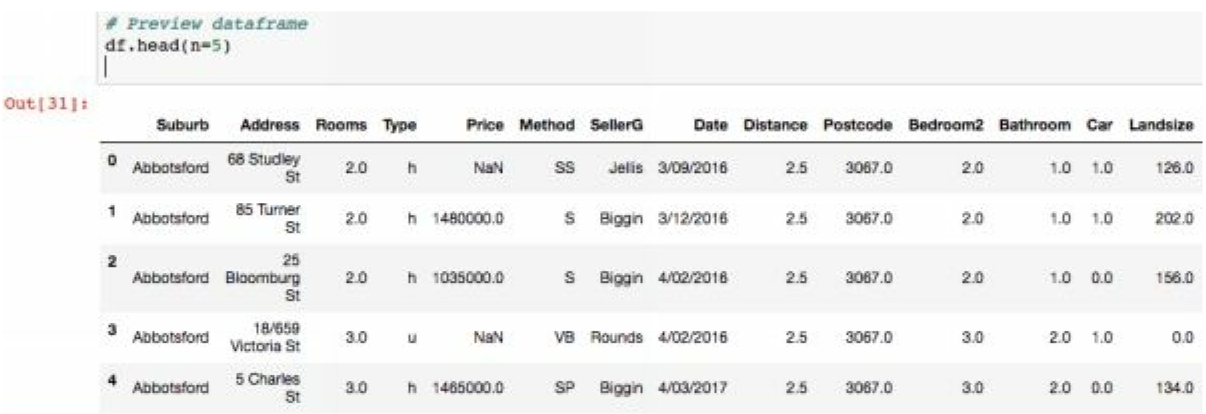
```
df = pd.read_csv('~/Desktop/Melbourne_housing_FULL-26-09-2017.csv')
```

In my case, I imported the dataset from my Downloads folder. As you move forward in machine learning and data science, it's important that you save datasets and projects in standalone and named folders for organized access. If you opt to save the .csv into the same folder as your Jupyter Notebook, you won't need to append a directory name or "~/."

Next, to preview the dataframe within Jupyter Notebook, enter the following command, with "n" representing the number of rows you wish to preview in relation to the head row.

```
df.head(n=5)
```

Right-click and select "Run" or navigate from the Jupyter Notebook menu: Cell > Run All



```
# Preview dataframe
df.head(n=5)
```

Out[31]:

	Suburb	Address	Rooms	Type	Price	Method	SellerG	Date	Distance	Postcode	Bedroom2	Bathroom	Car	Landsize
0	Abbotsford	68 Studley St	2.0	h	NaN	SS	Jellis	3/09/2016	2.5	3067.0	2.0	1.0	1.0	126.0
1	Abbotsford	85 Turner St	2.0	h	1480000.0	S	Biggin	3/12/2016	2.5	3067.0	2.0	1.0	1.0	202.0
2	Abbotsford	25 Bloomburg St	2.0	h	1035000.0	S	Biggin	4/02/2016	2.5	3067.0	2.0	1.0	0.0	156.0
3	Abbotsford	18/659 Victoria St	3.0	u	NaN	VB	Rounds	4/02/2016	2.5	3067.0	3.0	2.0	1.0	0.0
4	Abbotsford	5 Charles St	3.0	h	1465000.0	SP	Biggin	4/03/2017	2.5	3067.0	3.0	2.0	0.0	134.0

Figure 2: Previewing a dataframe in Jupyter Notebook

This will populate the dataset within Jupyter Notebook as shown in Figure 2. This step is not mandatory, but it is a useful technique for reviewing your dataset inside Jupyter Notebook.

3) Scrub the dataset

The next stage is to scrub the dataset. Remember, scrubbing is the process of refining your dataset. This involves modifying or removing incomplete, irrelevant or duplicated data. It may also entail converting text-based data to numerical values and the redesigning of features.

It is important to note that the scrubbing process can take place before or after importing the dataset into Jupyter Notebook. For example, the creator of the Melbourne Housing Market dataset has misspelled “Longitude” and “Latitude” in the head columns. As we will not be examining these two variables in our exercise, there is no need to make any changes. If, though, we did wish to include these two variables in our model, it would be prudent to first fix this error.

From a programming perspective, spelling mistakes in the column titles do not pose any problems as long as we apply the same keyword spelling to perform our commands. However, this misnaming of columns could lead to human errors, especially if you are sharing your code with team members. To avoid any potential confusion, it’s best to fix spelling mistakes and other simple errors in the source file before importing the dataset into Jupyter Notebook or another development environment. You can do this by opening the CSV file in Microsoft Excel (or equivalent program), editing the dataset, and then resaving it again as a CSV file.

While simple errors can be corrected within the source file, major structural changes to the dataset such as feature engineering are best performed in the development environment for added flexibility and to preserve the dataset for later use. For instance, in this exercise, we will be implementing feature engineering to remove a number of columns from the dataset, but we may later change our mind about which columns we wish to include. Manipulating the composition of the dataset in the development environment is less permanent and generally much simpler and quicker than doing so directly in the source file.

Scrubbing Process

Let’s first remove columns from the dataset that we don’t wish to include in the model by using the del df[' '] function and entering the vector (column) titles that we wish to remove.

```
# The misspellings of “longitude” and “latitude” are used, as the two misspellings were not corrected in the source file.
```

```
del df['Address']
del df['Method']
del df['SellerG']
del df['Date']
del df['Postcode']
del df['Lattitude']
del df['Longtitude']
del df['Regionname']
del df['Propertycount']
```

The Address, Regionname, and Propertycount columns were removed as property location is covered in other columns (Suburb and CouncilArea) and because we want to minimize non-numerical information (e.g. Address and Regionname). Postcode, Latitude, and Longitude were also removed because, again, property location is contained in the Suburb and CouncilArea columns. My assumption is that Suburb and CouncilArea tend to have more sway in buyers' minds than Postcode, Latitude, and Longitude—although Address deserves an honorable mention.

Method, SellerG, and Date were also removed because they were deemed to have less relevance in comparison to other variables. This is not to say that these variables don't impact property prices, rather the other eleven independent variables are sufficient for building a basic model. We can decide to add any of these variables into the model later, and you may choose to include them in your own model.

The remaining eleven independent variables (represented as X) in the dataset are Suburb, Rooms, Type, Distance, Bedroom2, Bathroom, Car, Landsize, BuildingArea, YearBuilt, and CouncilArea. The twelfth variable, located in the fifth column of the downloaded dataset, is the dependent variable, which is Price (represented as y). As mentioned, decision trees (including gradient boosting and random forests) are adept at managing large and high-dimensional datasets with a high number of variables.

The next step for scrubbing the dataset is to remove any missing values. Although there are numerous methods to manage missing values (e.g. calculating the mean, the median, or deleting missing values altogether), for this exercise, we want to keep it as simple as possible and we'll therefore not be examining rows with missing values. The obvious downside is that we have less data to analyze. As a beginner, it makes sense to master complete datasets before adding an extra dimension of difficulty in attempting to deal with missing values. Unfortunately, in the case of our sample dataset, we *do*

have a lot of missing values! Nonetheless, we still have ample rows available to proceed with building our model.

The following Pandas function can be used to remove rows with missing values:

```
df.dropna(axis=0, how='any', thresh=None, subset=None, inplace=True)
```

Keep in mind that it's important to drop rows with missing values after applying the del df function to remove columns (as shown in the previous step). This way, there's a better chance that more rows from the original dataset will be preserved. Imagine dropping a whole row because it was missing the value for a variable that would be later deleted like the post code in our model!

Next, let's convert columns that contain non-numerical data to numerical values using one-hot encoding. With Pandas, one-hot encoding can be performed using the get_dummies function:

```
features_df = pd.get_dummies(df, columns=['Suburb', 'CouncilArea', 'Type'])
```

This command converts column values for Suburb, CouncilArea, and Type into numerical values through the application of one-hot encoding.

Next, we need to remove the "Price" column because this column will act as our dependent variable (y) and for now we are only examining the eleven independent variables (X).

```
del features_df['Price']
```

Finally, create X and y arrays from the dataset using the matrix data type (as_matrix). The X array contains the independent variables and the y array contains the dependent variable of Price.

```
X = features_df.as_matrix()
y = df['Price'].as_matrix()
```

4) Split the dataset

We are now at the stage of splitting the data into training and test segments. For this exercise, we will proceed with a standard 70/30 split by calling the

Scikit-learn function below with an argument of “0.3.” The dataset’s rows are also shuffled randomly to avoid bias using the random_state function.

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)
```

5) Select the algorithm and configure its hyperparameters

As you will recall, we are using the gradient boosting algorithm for this exercise, as shown.

```
model = ensemble.GradientBoostingRegressor(
    n_estimators=150,
    learning_rate=0.1,
    max_depth=30,
    min_samples_split=4,
    min_samples_leaf=6,
    max_features=0.6,
    loss='huber'
)
```

The first line is the algorithm itself (gradient boosting) and comprises just one line of code. The lines below dictate the hyperparameters for this algorithm.

n_estimators represents how many decision trees to build. Remember that a high number of trees will generally improve accuracy (up to a certain point), but it will also increase the model's processing time. Above, I have selected 150 decision trees as an initial starting point.

learning_rate controls the rate at which additional decision trees influence the overall prediction. This effectively shrinks the contribution of each tree by the set learning_rate. Inserting a low rate here, such as 0.1, should improve accuracy.

max_depth defines the maximum number of layers (depth) for each decision tree. If "None" is selected, then nodes expand until all leaves are pure or until all leaves contain less than min_samples_leaf. Here, I have selected a high maximum number of layers (30), which will have a dramatic effect on the final result, as we will see later.

min_samples_split defines the minimum number of samples required to execute a new binary split. For example, min_samples_split = 10 means there must be ten available samples in order to create a new branch.

min_samples_leaf represents the minimum number of samples that must appear in each child node (leaf) before a new branch can be implemented. This helps to mitigate the impact of outliers and anomalies in the form of a low number of samples found in one leaf as a result of a binary split. For example, min_samples_leaf = 4 requires there to be at least four available

samples within each leaf for a new branch to be created.

max_features is the total number of features presented to the model when determining the best split. As mentioned in Chapter 11, random forests and gradient boosting restrict the total number of features shown to each individual tree to create multiple results that can be voted upon later.

If the max_features value is an integer (whole number), the model will consider max_features at each split (branch). If the value is a float (e.g. 0.6), then max_features is the percentage of total features randomly selected. Although max_features sets a maximum number of features to consider in identifying the best split, total features may exceed the max_features limit if no split can initially be made.

loss calculates the model's error rate. For this exercise, we are using huber which protects against outliers and anomalies. Alternative error rate options include ls (least squares regression), lad (least absolute deviations), and quantile (quantile regression). Huber is actually a combination of ls and lad.

To learn more about gradient boosting hyperparameters, you may refer to the Scikit-learn website:

http://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html

After imputing the model’s hyperparameters, we will implement Scikit-learn's fit function to start the model training process.

```
model.fit(X_train, y_train)
```

Lastly, we need to use Scikit-learn to save the training model as a file using the joblib.dump function, which was imported into Jupyter Notebook in Step 1. This will allow us to use the training model again in the future for predicting new real estate property values, without needing to rebuild the model from scratch.

```
joblib.dump(model, 'house_trained_model.pkl')
```

6) Evaluate the results

As mentioned earlier, for this exercise we will use mean absolute error to evaluate the accuracy of the model.

```
mse = mean_absolute_error(y_train, model.predict(X_train))
print ("Training Set Mean Absolute Error: %.2f" % mse)
```

Here, we input our y values, which represent the correct results from the

training dataset. The model.predict function is then called on the X training set and will generate a prediction with up to two decimal places. The mean absolute error function will then compare the difference between the model's expected predictions and the actual values. The same process is repeated with the test data.

```
mse = mean_absolute_error(y_test, model.predict(X_test))
print ("Test Set Mean Absolute Error: %.2f" % mse)
```

Let's now run the entire model by right-clicking and selecting "Run" or navigating from the Jupyter Notebook menu: Cell > Run All.

Wait a few seconds for the computer to process the training model. The results, as shown below, will then appear at the bottom of the notepad.

```
Training Set Mean Absolute Error: 27157.02
Test Set Mean Absolute Error: 169962.99
```

For this exercise, our training set mean absolute error is $27,157.02 and the test set mean absolute error is $169,962.99. This means that on average, the training set miscalculated the actual property value by a mere $27,157.02. However, the test set miscalculated by an average of $169,962.99.

This means that our training model was very accurate at predicting the actual value of properties contained in the training data. While $27,157.02 may seem like a lot of money, this average error value is low given the maximum range of our dataset is $8 million. As many of the properties in the dataset are in excess of seven figures ($1,000,000+), $27,157.02 constitutes a reasonably low error rate.

But how did the model fare with the test data? These results are less accurate. The test data provided less indicative predictions with an average error rate of $169,962.99. A high discrepancy between the training and test data is usually a key indicator of overfitting. As our model is tailored to the training data, it stumbled when predicting the test data, which probably contains new patterns that the model hasn't adjusted for. The test data, of course, is likely to contain slightly different patterns and new potential outliers and anomalies.

However, in this case, the difference between the training and test data is exacerbated by the fact that we configured the model to overfit the training data. An example of this issue was setting max_depth to "30." Although setting a high max_depth improves the chances of the model finding patterns

in the training data, it does tend to lead to overfitting. Another possible cause is a poor split of the training and test data, but for this model the data was randomized using Scikit-learn.

Lastly, please take into account that because the training and test data are shuffled randomly, your own results will differ slightly when replicating this model on your own machine.

14

MODEL OPTIMIZATION

In the previous chapter we built our first supervised learning model. We now want to improve its accuracy and reduce the effects of overfitting. A good place to start is modifying the model's hyperparameters.

Without changing any other hyperparameters, let's first start by modifying max_depth from "30" to "5." The model now generates the following results:

```
# Results will differ due to the randomized data split
Training Set Mean Absolute Error: 129412.51
```

Although the mean absolute error of the training set is higher, this helps reduce the problem of overfitting and should improve the results of the test data. Another step to optimize the model is to add more trees. If we set n_estimators to 250, we see this result:

```
# Results will differ as per the randomized data split
Training Set Mean Absolute Error: 118130.46
Test Set Mean Absolute Error: 159886.32
```

This second optimization reduces the training set's absolute error rate by approximately $11,000 and we now have a smaller gap between our training and test results for mean absolute error.

Together, these two optimizations underline the importance of maximizing and understanding the impact of individual hyperparameters. If you decide to replicate this supervised machine learning model at home, I recommend that you test modifying each of the hyperparameters individually and analyze their impact on mean absolute error. In addition, you will notice changes in the machine's processing time based on the hyperparameters selected. For instance, setting max_depth to "5" reduces total processing time compared to when it was set to "30" because the maximum number of branch layers are significantly less. Processing speed and resources will become an important consideration as you move on to working with larger datasets.

Another important optimization technique is feature selection. As you will

recall, we removed nine features while scrubbing our dataset. Now might be a good time to reconsider those features and analyze whether they have an effect on the overall accuracy of the model. “SellerG” would be an interesting feature to add to the model because the real estate company selling the property could have some impact on the final selling price.

Alternatively, dropping features from the current model may reduce processing time without having a significant effect on accuracy—or may even improve accuracy. To select features effectively, it is best to isolate feature modifications and analyze the results, rather than applying various changes at once.

While manual trial and error can be an effective technique to understand the impact of variable selection and hyperparameters, there are also automated techniques for model optimization, such as *grid search*. Grid search allows you to list a range of configurations you wish to test for each hyperparameter, and then methodically tests each of those possible hyperparameters. An automated voting process takes place to determine the optimal model. As the model must test each possible combination of hyperparameters, grid search does take a long time to run! Example code for grid search is shown at the end of this chapter.

Finally, if you wish to use a different supervised machine learning algorithm and not gradient boosting, much of the code used in this exercise can be replicated. For instance, the same code can be used to import a new dataset, preview the dataframe, remove features (columns), remove rows, split and shuffle the dataset, and evaluate mean absolute error.

http://scikit-learn.org is a great resource to learn more about other algorithms as well as the gradient boosting used in this exercise.

For a copy of the code, please contact the author at oliver.theobald@scatterplotpress.com or see the code example below. In addition, if you have troubles implementing the model using the code found in this book, please feel free to contact the author by email for extra assistance at no cost.

Code for the Optimized Model

```
# Import libraries
import pandas as pd
from sklearn.model_selection import train_test_split
```

```
from sklearn import ensemble
from sklearn.metrics import mean_absolute_error
from sklearn.externals import joblib

# Read in data from CSV
df =  pd.read_csv('~/Downloads/Melbourne_housing_FULL-26-09-2017.csv')

# Delete unneeded columns
del df['Address']
del df['Method']
del df['SellerG']
del df['Date']
del df['Postcode']
del df['Lattitude']
del df['Longtitude']
del df['Regionname']
del df['Propertycount']

# Remove rows with missing values
df.dropna(axis=0, how='any', thresh=None, subset=None, inplace=True)

# Convert non-numerical data using one-hot encoding
features_df = pd.get_dummies(df, columns=['Suburb', 'CouncilArea', 'Type'])

# Remove price
del features_df['Price']

# Create X and y arrays from the dataset
X = features_df.as_matrix()
y = df['Price'].as_matrix()

# Split data into test/train set (70/30 split) and shuffle
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)

# Set up algorithm
model = ensemble.GradientBoostingRegressor(
    n_estimators=250,
    learning_rate=0.1,
    max_depth=5,
    min_samples_split=4,
    min_samples_leaf=6,
    max_features=0.6,
    loss='huber'
)

# Run model on training data
model.fit(X_train, y_train)
```

```
# Save model to file
joblib.dump(model, 'trained_model.pkl')

# Check model accuracy (up to two decimal places)
mse = mean_absolute_error(y_train, model.predict(X_train))
print ("Training Set Mean Absolute Error: %.2f" % mse)

mse = mean_absolute_error(y_test, model.predict(X_test))
print ("Test Set Mean Absolute Error: %.2f" % mse)
```

Code for Grid Search Model

```
# Import libraries, including GridSearchCV
import pandas as pd
from sklearn.model_selection import train_test_split
from sklearn import ensemble
from sklearn.metrics import mean_absolute_error
from sklearn.externals import joblib
from sklearn.model_selection import GridSearchCV

# Read in data from CSV
df = pd.read_csv('~/Downloads/Melbourne_housing_FULL-26-09-2017.csv')

# Delete unneeded columns
del df['Address']
del df['Method']
del df['SellerG']
del df['Date']
del df['Postcode']
del df['Lattitude']
del df['Longtitude']
del df['Regionname']
del df['Propertycount']

# Remove rows with missing values
df.dropna(axis=0, how='any', thresh=None, subset=None, inplace=True)

# Convert non-numerical data using one-hot encoding
features_df = pd.get_dummies(df, columns=['Suburb', 'CouncilArea', 'Type'])

# Remove price
del features_df['Price']

# Create X and y arrays from the dataset
X = features_df.as_matrix()
y = df['Price'].as_matrix()

# Split data into test/train set (70/30 split) and shuffle
```

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=0)

# Input algorithm
model = ensemble.GradientBoostingRegressor()

# Set the configurations that you wish to test
param_grid = {
    'n_estimators': [300, 600, 1000],
    'max_depth': [7, 9, 11],
    'min_samples_split': [3, 4, 5],
    'min_samples_leaf': [5, 6, 7],
    'learning_rate': [0.01, 0.02, 0.6, 0.7],
    'max_features': [0.8, 0.9],
    'loss': ['ls', 'lad', 'huber']
}

# Define grid search. Run with four CPUs in parallel if applicable.
gs_cv = GridSearchCV(model, param_grid, n_jobs=4)

# Run grid search on training data
gs_cv.fit(X_train, y_train)

# Print optimal hyperparameters
print(gs_cv.best_params_)

# Check model accuracy (up to two decimal places)
mse = mean_absolute_error(y_train, gs_cv.predict(X_train))
print("Training Set Mean Absolute Error: %.2f" % mse)

mse = mean_absolute_error(y_test, gs_cv.predict(X_test))
print("Test Set Mean Absolute Error: %.2f" % mse)
```

BUG BOUNTY

Thank you for reading this absolute beginners' introduction to machine learning. While not customary practice in the publishing industry, we do offer a financial reward to readers for locating errors or bugs found in this book.

For this genre of writing—statistical-based data modeling—it is not uncommon for errors to emerge in the eye of the beholder. In other words, it's natural for readers to occasionally misinterpret diagrams, copy code incorrectly or misread important concepts. This is human nature, but to avoid readers attacking the author with a negative review and affecting future sales of this title, we invite you to report any bugs by first sending us an email at oliver.theobald@scatterplotpress.com

This way we can supply further explanations and examples over email to calibrate your understanding, or in cases where you're right and we're wrong, we offer a monetary reward of USD $20. This way you can make a tidy profit from your feedback and we can update the book to improve the standard of content for other readers.

FURTHER RESOURCES

This section lists relevant learning materials for readers that wish to progress further in the field of machine learning. Please note that certain details listed in this section, including prices, may be subject to change in the future.

| Machine Learning |

Machine Learning

Format: Coursera course
Presenter: Andrew Ng
Cost: Free
Suggested Audience: Beginners (especially those with a preference for MATLAB)
A free and well-taught introduction from Andrew Ng, one of the most influential figures in this field. This course has become a virtual rite of passage for anyone interested in machine learning.

Project 3: Reinforcement Learning

Format: Online blog tutorial
Author: EECS Berkeley
Suggested Audience: Upper intermediate to advanced
A practical demonstration of reinforcement learning, and Q-learning specifically, explained through the game Pac-Man.

| Basic Algorithms |

Machine Learning With Random Forests And Decision Trees: A Visual Guide For Beginners

Format: E-book
Author: Scott Hartshorn
Suggested Audience: Established beginners

A short, affordable (USD $3.20), and engaging read on decision trees and random forests with detailed visual examples, useful practical tips, and clear instructions.

Linear Regression And Correlation: A Beginner's Guide
Format: E-book
Author: Scott Hartshorn
Suggested Audience: All
A well-explained and affordable (USD $3.20) introduction to linear regression, as well as correlation.

| The Future of AI |

The Inevitable: Understanding the 12 Technological Forces That Will Shape Our Future
Format: E-Book, Book, Audiobook
Author: Kevin Kelly
Suggested Audience: All (with an interest in the future)
A well-researched look into the future with a major focus on AI and machine learning by The New York Times Best Seller Kevin Kelly. Provides a guide to twelve technological imperatives that will shape the next thirty years.

Homo Deus: A Brief History of Tomorrow
Format: E-Book, Book, Audiobook
Author: Yuval Noah Harari
Suggested Audience: All (with an interest in the future)
As a follow-up title to the success of *Sapiens: A Brief History of Mankind,* Yuval Noah Harari examines the possibilities of the future with notable sections of the book examining machine consciousness, applications in AI, and the immense power of data and algorithms.

| Programming |

Learning Python, 5th Edition

Format: E-Book, Book
Author: Mark Lutz
Suggested Audience: All (with an interest in learning Python)
A comprehensive introduction to Python published by O'Reilly Media.

Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems
Format: E-Book, Book
Author: Aurélien Géron
Suggested Audience: All (with an interest in programming in Python, Scikit-Learn and TensorFlow)
As a highly popular O'Reilly Media book written by machine learning consultant Aurélien Géron, this is an excellent advanced resource for anyone with a solid foundation of machine learning and computer programming.

| Recommendation Systems |

The Netflix Prize and Production Machine Learning Systems: An Insider Look
Format: Blog
Author: Mathworks
Suggested Audience: All
A very interesting blog article demonstrating how Netflix applies machine learning to form movie recommendations.

Recommender Systems
Format: Coursera course
Presenter: The University of Minnesota
Cost: Free 7-day trial or included with $49 USD Coursera subscription
Suggested Audience: All
Taught by the University of Minnesota, this Coursera specialization covers fundamental recommender system techniques including content-based and collaborative filtering as well as non-personalized and project-association recommender systems.
.

| Deep Learning |

Deep Learning Simplified
Format: Blog
Channel: DeepLearning.TV
Suggested Audience: All
A short video series to get you up to speed with deep learning. Available for free on YouTube.

Deep Learning Specialization: Master Deep Learning, and Break into AI
Format: Coursera course
Presenter: deeplearning.ai and NVIDIA
Cost: Free 7-day trial or included with $49 USD Coursera subscription
Suggested Audience: Intermediate to advanced (with experience in Python)
A robust curriculum for those wishing to learn how to build neural networks in Python and TensorFlow, as well as career advice, and how deep learning theory applies to industry.

Deep Learning Nanodegree
Format: Udacity course
Presenter: Udacity
Cost: $599 USD
Suggested Audience: Upper beginner to advanced, with basic experience in Python
Comprehensive and practical introduction to convolutional neural networks, recurrent neural networks, and deep reinforcement learning taught online over a four-month period. Practical components include building a dog breed classifier, generating TV scripts, generating faces, and teaching a quadcopter how to fly.

| Future Careers |

Will a Robot Take My Job?
Format: Online article

Author: The BBC
Suggested Audience: All
Check how safe your job is in the AI era leading up to the year 2035.

So You Wanna Be a Data Scientist? A Guide to 2015's Hottest Profession
Format: Blog
Author: Todd Wasserman
Suggested Audience: All
Excellent insight into becoming a data scientist.

The Data Science Venn Diagram
Format: Blog
Author: Drew Conway
Suggested Audience: All
The popular 2010 data science diagram designed by Drew Conway.

DOWNLOADING DATASETS

Before you can start practicing algorithms and building machine learning models, you will first need data. For beginners starting out in machine learning, there are a number of options. One is to source your own dataset from writing a web crawler in Python or utilizing a click-and-drag tool such as Import.io to crawl the Internet. However, the easiest and best option to get started is by visiting kaggle.com.

As mentioned throughout this book, Kaggle offers free datasets for download. This saves you the time and effort of sourcing and formatting your own dataset. Meanwhile, you also have the opportunity to discuss and problem-solve with other users on the forum, join competitions, and simply hang out and talk about data.

Bear in mind, however, that datasets you download from Kaggle will inherently need some refining (through scrubbing) to tailor to the machine learning model that you decide to build. Below are four free sample datasets from Kaggle that may prove useful to your further learning in this field.

World Happiness Report

What countries rank the highest in overall happiness? Which factors contribute most to happiness? How did country rankings change between the 2015 and 2016 reports? Did any country experience a significant increase or decrease in happiness? These are the questions you can ask of this dataset recording happiness scores and rankings using data from the Gallup World Poll. The scores are based on answers to the main life evaluation questions asked in the poll.

Hotel Reviews

Does having a five-star reputation lead to more disgruntled guests, and conversely, can two-star hotels rock the guest ratings by setting low expectations and over-delivering? Or are one and two-star rated hotels simply rated low for a reason? Find all this out from this sample dataset of hotel reviews. This particular dataset covers 1,000 hotels and includes hotel name, location, review date, text, title, username, and rating. The dataset is sourced from the Datafiniti's Business Database, which includes almost every hotel in

the world.

Craft Beers Dataset

Do you like craft beer? This dataset contains a list of 2,410 American craft beers and 510 breweries collected in January 2017 from CraftCans.com. Drinking and data crunching is perfectly legal.

Brazil's House of Deputies Reimbursements

As politicians in Brazil are entitled to receive refunds from money spent on activities to “better serve the people,” there are interesting findings and suspicious outliers to be found in this dataset. Data on these expenses are publicly available, but there is very little monitoring of expenses in Brazil. So don’t be surprised to see one public servant racking up over 800 flights in twelve months, and another that recorded R 140,000 (USD $44,500) on post expenses—yes, snail mail!

FINAL WORD

Thank you for purchasing this book. You now have a baseline understanding of the key concepts in machine learning and are ready to tackle this challenging subject in earnest. This includes learning the vital programming component of machine learning.

To further your study of machine learning, I strongly recommend that you enroll in the free Andrew Ng Machine Learning course offered on Coursera.

If you have any direct feedback, both positive and negative, or suggestions to improve this book, please feel free to send me an email at oliver.theobald@scatterplotpress.com. This feedback is highly valued and I look forward to hearing from you.

Finally, I would like to express my gratitude to my colleagues Jeremy Pederson and Rui Xiong for their assistance in kindly sharing practical machine learning tips and some code used in this book.

Thank you,
Oliver Theobald

---

[1] BBC, Will A Robot Take My Job?, 2015, http://www.bbc.com/news/technology-34066941

[2] Nearshore Americas, Machine Learning Adoption Thwarted by Lack of Skills and Understanding, 2017, http://www.nearshoreamericas.com

[3] Arthur Samuel, Some Studies in Machine Learning Using the Game of Checkers*,* IBM Journal of Research and Development, Vol. 3, Issue. 3, 1959.

[4] Arthur Samuel, Some Studies in Machine Learning Using the Game of Checkers*,* IBM Journal of Research and Development, Vol. 3, Issue. 3, 1959.

[5] DataVisor, Unsupervised Machine Learning Engine, 2017, https://www.datavisor.com/unsupervised-machine-learning-engine/

[6] Kevin Kelly, The Inevitable: Understanding the 12 Technological Forces That Will Shape Our Future, Penguin Books, 2016.

[7] Torch, What is Torch? http://torch.ch/, 2017